F# ile Fonksiyonel Programlama

Ali Özgür

İÇİNDEKİLER

BÖLÜM 1: GİRİŞ **1**

F# ile Tanışma 2

printfn Fonksiyonu 7

Kısa F# Tarihçesi 9

Neden F#? 12

Az Seremonili Söz Dizimi 12

Sade ve Şık Tip Tanımları 13

Güçlü Tip Sistemi 14

Sade ve Yetenekli Veri Yapıları 16

Eş Zamanlı ve Paralel İşlemler 19

Fonksiyonel Olmayan Yöntem Desteği 22

Geniş Uygulama Yelpazesi 23

Aktif Geliştirici Topluluğu 23

Hazır Paketler 23

Fonksiyonlara Matematiksel Bakış 24

Fonksiyonların İlginç Özellikleri 26

Değerleme Sırası Önemli Mi Değil Mi? 28

Fonksiyonların İlginç Olmayan Özellikleri 30

Fonksiyonların Parametre Sayısı 31

Fonksiyonel Programlama Nedir? 32

Bildirimsel ve Yordamsal Programlama Yaklaşımları 34

Neler Öğrendik? 37

Bölüm 2: F# Geliştirme Platformu 39

Geliştirme Araçları 40

Derleyici ve Yorumlayıcı Kavramları 41

F# Derleyicisi 41

F# Etkileşimli Ortamı 43

Merhaba F# 46

.NET Core Kurulumu 46

Visual Studio Code Kurulumu 47

F# Standart Dosya Uzantıları 54

Derleyici ve Etkileşimli Ortam Değişkenleri 55

Neler Öğrendik? 55

Bölüm 3: F# Temelleri 57

Söz Dizimi Kuralları 58

Girinti Kullanımı (Indentation) 58

let Anahtar Kelimesi 60

do Anahtar Kelimesi 63

Yorum Satırları 64

Koşullu Derleme 65

Tanımlayıcı ve Anahtar Kelimeler 66

Shebang 68

Basit Veri Tipleri 69

Tipler Arası Dönüşüm 74

Karşılaştırma ve Eşitlik 77

Bit Manipülasyonu 78

Mantıksal/Lojik Değerler 79

Karakterler 79

Metinler 81

Fonksiyonlar 82
Fonksiyonların İmzası 85
Parametresiz Fonksiyon Tanımları 88
İsimsiz/Anonim Fonksiyonlar (Lambda İfadeleri) 90
Fonksiyonların İleri Seviye Kullanımı 91
Fonksiyon İçinde Fonksiyon Tanımı 91
Currying 92
Curried Fonksiyonların İmzası 94
Fonksiyonlarda Kısmi Uygulama Yöntemi 95
Kısmi Uygulama Uyumlu Fonksiyon Tasarımı 96
Öz Yinelemeli Fonksiyonlar 99
Döngü Yapıları Olarak Öz Yinelemeli Fonksiyon Kullanımı 102
Karşılıklı Öz Yinelemeli Fonksiyonlar 105
Temel Veri Tipleri 107
Unit 107
Null 109
Tuple (Değer Grubu) 111
Option (Opsiyon) 114
Yapısal Eşitlik 117
Kod Organizasyonu 123
Çözümler ve Projeler 123
Dosya Sıralaması 125
Modüller ve Alan Adları 129
Tip ve Fonksiyonların Organzasyonu 133
Neler Öğrendik? 135

Bölüm 4: Fonksiyonel Programlama 137

Desen Eşleme (Pattern Matching) 138

Desen Eşleme Tipleri 144

function Anahtar Kelimesi ile Desen Eşleme 149

try...with ile Desen Eşleme 150

Kayıtlar (Record) 151

Kayıtları Parçalama ve Alan Değerlerini Sökme 155

Kayıtları Güncelleme 156

Ayrışık Bileşim (Discrıminated Union) 157

Kümelerin Bileşimini Alma 158

Etiket İsimlendirme ve Tip Tanımlama İpuçları 161

Karşılıklı Öz Yinelemeli Ayrışık Bileşimler 162

Desen Eşleme 163

Enum Tipler 164

Alıştırma - 2 164

Tiplere Davranış Ekleme 166

Statik Üyeler 170

Standart Tipleri İçin Üye Tanımlama 171

Tip Genelleme (Generics) 174

Doğrudan Genelleme 174

Dolaylı Genelleme 177

Sonradan Değerleme (Lazy Evaluation) 178

Neler Öğrendik? 182

Bölüm 5: Koleksiyonlar 185

Liste (List) 186

Eleman Ekleme Operatörü :: 188

Birleştirme Operatörü @ 189

İfadeler ile Liste Oluşturma (List Comprehension) 189

Eşitlik 192

Dizi (Array) 192

Kesit Alma 196

Çok Boyutlu Diziler (Multi-Dimensional Arrays) 196

Düzensiz Diziler (Jagged Arrays) 199

Eşitlik 199

Sekans (Sequence) 200

Yield! (Yield Bang) 202

Küme (Set) 204

Anahtar Değer Haritası (Map) 207

List Modülü 208

Sorgu İfadeleri (Query Expressions) 220

Neler Öğrendik? 227

Bölüm 6: Genel Amaçlı Programlama 229

Değişkenlere Gerçekten İhtiyacımız Var Mı? 230

Değer Tipleri ve Referans Tipleri 235

Değişkenler 240

byref Fonksiyon Parametreleri 245

.NET Koleksiyonları 246

List 247

Dictionary 248

HashSet 249

Döngü Yapıları (For ve While) 250
Koşullu Dallanma Yapıları (If/Else) 254
İstisnalar (Exceptions) 256
İstisna Tipi Oluşturma 257
İstisna Fırlatma 258
İstisna Yakalama 260
Ölçü Birimleri 264
Neler Öğrendik? 268

Bölüm 7: Nesne Yönelimli Programlama 271

Nesne Yönelimli Programlama Nedir? 272
Sınıf Tanımlama 272
Varsayılan Kurucu Fonksiyonlar 273
İlklendirme 274
Statik İlklendirme 275
Statik Değer, Değişken ve Fonksiyonlar 278
İlave Kurucu Fonksiyonlar 279
Varsayılan Kurucusu Olmayan Sınıflar 280
Sınıfların Organizasyonu 281
Karşılıklı Öz Yinelemeli Sınıflar 283
Üye Özellikler 284
val ile Üye Özellik Tanımlama 286
val ile Alan Tanımlama 287
Statik Üye Özellikler 288
Üye Metodlar 289
Parametre Tanım Yöntemleri 290
Parametresiz Metodlar 291
Öz Yinelemeli Üye Metodlar 292

Üye Erişim Kısıtlayıcıları 293
Kalıtım 296
Soyut Sınıflar 297
Geçersiz Kılma 299
Soyut Sınıf Kullanımı 299
Kontratlar/Ara Birimler 302
Sınıflar Arası Tip Dönüşümü 306
Üst Tipe Dönüşüm (Upcasting) 306
Alt tipe dönüşüm (Downcasting) 308
IDisposable Kullanım Desenleri 309
Neler Öğrendik? 315

Bölüm 8: Gelişmiş Fonksiyonel Programlama Yöntemleri 317

Aktif Desenler 318
Tek Durumlu Aktif Desenler 320
Kısmi Aktif Desenler 321
Parametreli Aktif Desenler 322
Çok Durumlu Aktif Desenler 323
Aktif Desen Kullanım Şablonları 324
Kuyruk Özyenilemeli Fonksiyonlar 325
Biriktirici Desen (Accumulator Pattern) 326
Uzantılar Deseni (Contınuations Pattern) 327
Aktarım Operatörleri 327
Fonksiyon Kompozisyonu 332
Monadlar ve Hesaplama İfadeleri 334
Monad 334
Kural 338
Kural 339
Hesaplama İfadeleri 340

İpuçları 347
Herşeyi Parametrize Et 347
Strateji Deseni 348
Dekoratör Deseni 349
Kısmi Uygulama 351
Uzantılar ile Kontrolü Devretme 355
Uzantılar ile Zincirleme 356
Neler Öğrendik? 357

Bölüm 9: Asenkron ve Paralel Programlama 359

Giriş ve Temel Kavramlar 360
Thread Sınıfı 363
Auto ve Manual Resent Event 366
Thread'leri Bekleme 369
.NET Thread Havuzu 372
Paylaşılan Veriyi Okuma ve Güncelleme 373
Karşılıklı Kilitleme 378
Asenkron İş Akışları 379
Async İfadeler 380
Asenkron Akışları Başlatma 383
Parametresiz Asenkron İfadeler 387
Birden Fazla Asenkron İşlemi Başlatmak 388
İstisnalar 389
Asenkron İşlemleri İptal Etme 391
Kısıtlamalar ve Performans 393

Task Sınıfı ve Paralel Programlama 394
parallel.For ve Parallel.ForEach 395
Array.Parallel Modülü 397
Task<'T> ile Paralel İşlemler 397
İşlemlerin İptali 399
İstisnaların Yakalanması 400
MailboxProcessor Sınıfı 401
Neler Öğrendik? 409
Terimler 410

1

Giriş

Bu Bölümde

F# ile Tanışma	2
Kısa F# Tarihçesi	9
Neden F#?	12
Fonksiyonlara Matematiksel Bakış	24
Fonksiyonların İlginç Özellikleri	26
Fonksiyonel Programlama Nedir?	32
Neler Öğrendik?	37

Bu bölümde önce F#'ın kısa tarihçesine göz atarak, Neden F#? ve F# Programlama Dili neye benzer? sorularının cevaplarını vereceğiz. Matematiksel anlamda fonksiyonları ve fonksiyonların bazı ilginç özelliklerini ele aldıktan sonra da fonksiyonel programlamanın tanımı ile bölümü tamamlayacağız.

F# İLE TANIŞMA

Programlama dili kitapları ekrana **Merhaba Dünya!** yazdırmak için kullanılan kod ile başlar. Bu klasik kod parçası öğrenmek üzere olduğumuz dil hakkında fikir edinmemizi sağlar. Biz de kitabımıza bu klasik ile başlıyoruz.

```
printfn "F#'dan Merhaba Dünya!"
```

F# ile terminale bir şeyler yazdırmak işte bu kadar kısa ve basit. Aşağıda günlük çalışmanızda sık sık karşılaşacağımız F# yapılarının büyük çoğunluğu kısa örnekler şeklinde verilmiştir. Yapılan işin karmaşıklığı ve ayrıntısı artsa bile F#'ın sadeliğinden ödün vermediğini ve kodun oldukça şık olduğunu örneklere bakarak rahatlıkla söyleyebiliriz. Yazılan kodun şıklığı, estetik görünüme ilave olarak kolay okunan, kolay anlaşılan ve kolay yönetilebilen kodu simgeler.

```
(* 01_0_03.fsx *)

// tek satırlık yorumlar için // kullanılır
(*
    Birden fazla satırlı yorumlar için
    (* *) çifti kullanılır
*)

// "let" anahtar kelimesi ile değeri
// değiştirilemeyen (immutable) ifadeler tanımlanır
let sayı = 5
let ondalıkSayı = 3.14
let metin = "Merhaba Dünya!"

// İfadeleri `` `` arasında yazarak
// F# anahtar kelimelerini de ifade adı
// olarak kullanabilirsiniz.
let ``let``= "F# ile Fonksiyonel Programlama"

// `` `` kullanarak boşluk içeren ifade isimleri
// oluşturulabilir. Bu kullanım özellikle
// birim testlerin (unit test) fonksiyon isimlerinde
// oldukça faydalı olacaktır.
let ``Cümle gibi değer``="Cümle gibi değer ifadesinin değeri"

// F# değer ifadelerinin ismi olarak
// UTF-8 karakterleri kullanılmasına izin verir.
let çÇşŞğĞüÜöÖİı = "Türkçe'ye özel karakterler"
```

Aşağıdaki örnek kod parçasında farklı tipteki değerlerin yer tutucular kullanılarak terminale nasıl yazdırıldığını inceleyebilirsiniz.

```
printf "Sayı değeri %d" "1"  // Hata, %d tam sayı tipi yer tutucusu
printf "Sayı değeri %d" 1    // Doğru

printfn "Sayı değeri %f" 1   // Hata, %f ondalık sayı tipi yer tutucusu
printfn "Sayı değeri %f" 1.0 // Doğru

printfn "Harf değeri %c" "A" // Hata, %c karakter tipi yer tutucusu
printfn "Harf değeri %c" 'A' // Doğru

printf "Sayı değerileri %d ve %d" 1   // Hata, iki değer bekleniyor
printf "Sayı değerileri %d ve %d" 1 2 // Doğru
```

NOT

Terminale metin yazdırmak için **.NET** standart kütüphanesindeki `Console` sınıfının `Write` ve `WriteLine` statik metodları da kullanılabilir. Ancak, F#'ın `printf`, `printfn` ve `sprintf` fonksiyonları tip uyumluluğu ve değer adetlerini kontrol ettiği için daha güvenlidirler.

Kısa F# Tarihçesi

F#, Türkçe **efşarp** olarak telafuz edilen, yabancı kaynaklarda **FSharp** olarak da rastlayabileceğiniz, yordamsal (imperative) ve bildirimsel (declarative) programlama yaklaşımlarının her ikisini de destekleyen çok yönlü (multi paradigm) ve fonksyionel bir programlama dilidir.

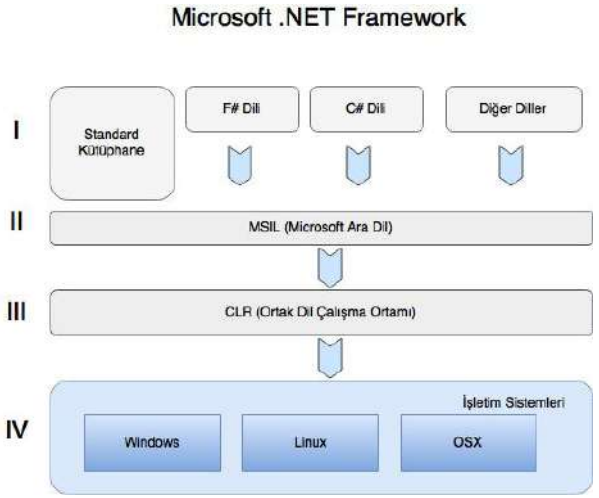
DİKKAT

Fonksiyionel programlama dili ifadesindeki fonksiyonel ibaresi ilk etapta **çok faydalı, işe yarayan** benzeri anlamlar çağırıştırsa da kitapta bu anlamlarda kullanılmamıştır. **Fonksiyonel programlama** programlama dilleri sınıflandırmasında matematikteki fonksiyonları ve özelliklerini temel alan yaklaşımı ifade eder.

F#, Microsoft tarafından tasarlanıp geliştirilen açık kaynak kodlu bir dilidir. F#'ın geliştirilmesindeki temel motivasyon Microsoft'un en önemli platformlarından biri olan **.NET'in** tasarım prensiplerine kadar uzanır. .NET, diller, derleyiciler, standart kütüphaneler ve sanal çalışma ortamı gibi yazılım geliştirme ve bu yazılımların çalıştırıldığı bileşenleri içeren bir yapıdır. .NET'i destekleyen programlama dilleri ile geliştirilmiş programlar dillerin kendilerine özel derleyicileri tarafında derlenir. Derleyiciler tarafından **MSIL** (Microsoft Intermediate Language) olarak isimlendirilen ara bir dile dönüştürülen programlar ortak dil çalışma ortamı olan **CLR** (Common Language Runtime) tarafından çalıştırılabilir.

MSIL, işletim sistemi ve CPU mimarisi bağımsız bir dildir. .NET'i destekleyen programalama dillerinin (C#, VB.NET ve F#) derleyicileri MSIL kodu üretirler. Genelde MSIL kodu elle yazılmaz. .NET'i destekleyen herhangi bir dil ile geliştirilen ve MSIL'e derlenen programlar Windows, Linux ve OSX işletim sistemleri üzerinde CLR içinde çalıştırılabilir.

.NET ilk çıktığında geliştirici araçları ve CLR bileşenleri sadece Windows işletim sistemde çalışıyodu. Kısa bir süre sonra bağımsız bir grup yazılımcı Linux ve OSX'de de çalışabilen Mono isimli açık kaynak bir .NET versiyonu geliştirdi. 2015 yılına kadar Mono lisanslama koşulları nedeni ile Microsoft'un orjinal kodunu kullanmadı. Ancak, 2015 yılı itibariyle Microsoft da Mono'ya doğrudan kod katkısı sağlamaya başlamıştır. Buna ilave olarak Microsoft Windows, Linux ve OSX'de çalışan ve .NET Core olarak adlandırılan yeni bir .NET versiyonu geliştirmektedir. 2017 yılında .NET Core 2.0 dağıtımı kullanıma sunulmuştur.



Ancak, switch/case benzeri yapılar yazım açısından zahmetli olup genellemeye uygun değildirler. Tanım kümesinin tüm elemanlarının switch/case ile değer kümesinden bir eleman ile eşleştirilmesi pratik olarak mümkün değildir. Bu nedenle fonksiyonları, bir hesaplama yaptığı izlenimine kapılmamıza da neden olan, aşağıdaki şekilde yazarak genelleştirebiliriz.

```
(* 01_2_02.fsx *)
let f (x) = x * x
```

Fonksiyonların ikinci ilginç özelliği ise yan etkilerinin olmamasıdır. Yan etki fonksiyonun eşleştirme dönüşümünü yaparken girdi olarak verilen tanım kümesindeki değeri de değiştirmesi durumuna denir.

Örneğin `f(x) = x * x` fonksiyonuna girdi olarak verilen değer kümesindeki `x = 5` değerinin `y = f 5` ifadesi ile yapılan dönüşüm işlemi sonrasında hala 5'e eşit olması f(x) fonksiyonunun yan etkisi olmadığını gösterir.

```
(* 01_2_03.fsx *)
let f(x) = x * x

let x = 5
let y = f 5

printfn "x = %d" x
printfn "y = %d" y
```

Bu iki özelliği sağlayan fonksiyonları matematikçiler ve fonksiyonel programcılar saf fonksiyonlar olarak adlandırır. Saf fonksiyonlar aynı girdi değerleri için her zaman aynı çıktıyı üretir ve bu işlem sonsuza dek farklı değerler ile tekrarlansa bile fonksiyonun davranışı değişmez. İlave olarak saf fonksiyonlar hiç bir zaman girdinin değerini değiştirmez. Saf fonksiyonlar fonksiyonel programlama çerçevesinde aşağıdaki yöntemlerin uygulanmasını mümkün kılar.

» Örneğin 100 çekirdekli bir işlemciniz varsa 1 ile 100 arasındaki sayıların karelerini almayı eş zamanlı ve paralel olarak her çekirdekte bir işlem yapılacak şekilde kodlayabilirsiniz. Fonksiyonların birinci özelliği eş zamanlı ve paralel çağırıları mümkün kılar.

2

F# Geliştirme Platformu

Bu Bölümde

Geliştirme Araçları	40
Derleyici ve Yorumlayıcı Kavramları	41
F# Derleyicisi	41
F# Etkileşimli Ortamı	43
Merhaba F#	46
F# Standart Dosya Uzantıları	54
Derleyici ve Etkileşimli Ortam Değişkenleri	55
Neler Öğrendik?	55

Bu bölümde, F# ile geliştirilen kodu çalıştırmak ve derlemek için kullanılan FSI ve FSC bileşenlerini ele alıyoruz. Bu iki temel bileşenin kullanımını öğrendikten sonra da klaskik "Merhaba Dünya!" programını F# ile oluşturup bölümü tamamlıyoruz.

GELİŞTİRME ARAÇLARI

F# ile Windows, Linux ve OSX işletim sistemleri üzerinde aşağıdaki tabloda verilen editörleri kullanarak programlama yapabilirsiniz. Alternatif olarak sadece F# derleyicisini ve işletim sisteminize uygun .NET versiyonunu kurarak herhangi bir metin editörü ile de kod yazabilirsiniz.

Editör	İşletim Sistemi	Lisanslama	Editör Versiyonu	.NET
Visual Studio	Windows	Ücretsiz Community Edition mevcut	2012, 2013, 2015 ve 2017	.NET Framework, .NET Core
Visual Studio Code	Ücretsiz	Windows, Linux, OSX	-	.NET Framework, .NET Core, Mono
Visual Studio for Mac	Ücretsiz Community Edition mevcut	OSX	-	.NET Core, Mono
JetBrains Rider	Ücretli	Windows, Linux, OSX	-	.NET Core
MonoDevelop	Ücretsiz	Windows, Linux, OSX	-	Mono

İşletim sistemi bazında F# derleyicisi, .NET ve editör kurulumu ayrıntılarına ***fsharp.org*** web sistesindeki **Use** linkini kullanarak ulaşabilirsiniz.

İPUCU Kitaptaki örneklerin çoğunu herhangi bir kurulum yapmadan ***glot.io*** web sistesini kullanarak online olarak çalıştırabilirsiniz.

F# geliştirme bileşenleri açık kaynaklıdır ve bağımsız geliştirici topluluğu tarafından desteklenir. Ancak, Microsoft Windows üzerinde Visual Studio ile birlikte gelen F# derleyicisini ve etkileşimli ortamını açık kaynak araçlardan farklı ve ayrı bir paket olarak dağıtır. Her iki dağıtım da ücretsizdir ve komut satırında kullanılan derleyici ve yorumlayıcı komut isimleri dışında iki dağıtım arasında bizi etkileyecek ciddi bir fark yoktur.

» Windows üzerindeki Microsoft dağıtımında derleyici komutu `fsc`, etkileşimli ortam komutu ise `fsi`'dir.

» Linux ve OSX üzerindeki açık kaynak F# araçlarının derleyici komutu `fsharpc`, etkileşimli ortam komutu ise `fsharpi`'dir.

Derleyici ve Yorumlayıcı Kavramları

Herhangi bir programlama dilinde geliştirme yapmak için ihtiyaç duyulan en basit bileşen **derleyicidir**. Derleyici metin dosyası olarak yazılan kodun dil kurallarına göre denetlenmesi ve optimize edilmesinden sonra çalıştırılabilir programın üretilmesini sağlar. **Örneğin;** C#, C++, Go ve Java dilleri derlenen dillerdir. Bazı programlama dilleri derleyici yerine **yorumlayıcı** adı verilen bileşeni kullanır. Yorumlayıcı derleyiciden farklı olarak kodun çalışma anında yorumlanarak çalıştırılmasını sağlar. **Örneğin;** JavaScript, Python ve PHP gibi dillerin kodu çalışma anında yorumlayıcılar tarafından yorumlanır.

F#, hem derleyicisi hem de yorumlayıcısı olan programlama dillerinden birisidir. Çalıştırılabilir bir program üretmek için F# derleyicisi kullanılırken, kod dosyalarının içindeki kodun yorumlanarak etkileşimli olarak çalıştırılması için yorumlayıcı kullanılır.

Gelin şimdi bu iki bileşeni nasıl kullanacağımızı görelim.

F# Derleyicisi

F# derleyicisi açık kaynak kodlu bir bileşendir. F# derleyicisini Windows, Linux ve OS X işletim sistemlerine **F# Geliştirme Araçları** kurulum paketlerini indirip kurabilirsiniz. F# derleyicisini komut satırından `fsc` (Windows üzerinde) komutu ile kullanabilirsiniz. Şimdi gelin basit bir konsol uygulaması kodunu `fsc` komutunu kullanarak derleyelim ve uygulamamızı çalıştıralım.

```
(* 02_1_01.fs *)
[<EntryPoint>]
let main args =

    // Ekrana yaz
    printfn "Merhaba Dünya!"

    // Geçilen parametreleri sırasıyla ekrana bas
    args |> Array.iter( fun s -> printfn "Merhaba %s." s)

    printfn "-------"
    printfn "Sonlandırmak için lütfen ENTER'a basın."
    let l = System.Console.ReadLine()

    0
```

3

F# Temelleri

Bu Bölümde

Söz Dizimi Kuralları	58
Basit Veri Tipleri	69
Fonksiyonlar	82
Fonksiyonların İleri Seviye Kullanımı	91
Temel Veri Tipleri	107
Yapısal Eşitlik	117
Kod Organizasyonu	123
Neler Öğrendik?	135

Bu bölümde, ilk olarak F#'ın söz dizimi kurallarını inceleyecek, daha sonra basit (int, string, bool) ve temel veri tiplerini (değer grubu, unit, listeler, diziler) ele alarak F#'ın yapı taşları olan fonksiyonların ayrıntılarına bakacağız.

Son olarak da faydalı kod organizasyonu ipuçları ile bölümü tamamlayacağız.

Söz Dizimi Kuralları

F#, göze hoş gelen, okunması kolay ve kodun çalışmasına doğrudan etkisi olmayan fazlalıklardan arındırılmış bir söz dizimine sahiptir. F# söz dizimi sade olmakla birlikte oldukça şıktır ve farklı dil yapılarını güzel bir şekilde ifade etmenizi sağlar.

Gelin şimdi F# söz diziminin temelini oluşturan kavram ve kuralları inceleyelim.

Girinti Kullanımı (Indentation)

F#'da kod blokları, ya da daha doğru tabirle kod alanları (scope), girintiler (indentation) ile birbirinden ayrılır. Girintiler her zaman 4 karakter uzunluğundaki boşluklar ile verilir ve TAB özel karekteri kullanılmaz. Ancak, tüm kod editörleri TAB tuşuna basınca TAB karakteri yerine belirli sayıda boşluk karakteri basacak şekilde ayarlanabilir. Bu nedenle pratikte TAB tuşunu kullanmanızın önünde bir engel yoktur.

C, C++, C#, Java ve JavaScript gibi dillerde kod alanlarını belirlemek için süslü parantez olarak adlanrılılan {} karakter çifti kullanılırken F#'da özel bir karakter veya karakter kullanılmaz. Kod alanlarının hangi satırda başlayıp hangi satırda bittiği girintiler ile belirlendiği için bitiş işaretçisi olarak **noktalı virgül** (;) veya farklı karakterler kullanılmaz.

```
(* 03_1_01.0.fsx*)

let sayı = 42

// Global alanda Modül tanımı
module Modül1 =
    // Girinti kullanıldığı için Modül1 alanına aittir
    let sayı' = 43
    let kırkÜçEkle x = sayı' + x

// Girinti yok o nedenle Global alana ait
let sayı'' = 44

// kırkÜçEkle fonksiyonunun önüne Modül1 ekleyerek
// Global alandan kullanabiliriz
Modül1.kırkÜçEkle 44

// Global alanda fonksiyon tanımı
let birArttırVeKaresiniAl x =
    // Fonksiyon alanı başlangıcı
```

```
    let t = x + 1
    t * t
    // Fonksiyon alanı bitişi

// sayı'' değer ve birArttırVeKaresiniAl fonksiyonu
// her ikisi de Global alanda
birArttırVeKaresiniAl sayı''

// Global alanda fonksiyon tanımı
let çiftMiTekMi x =
    // Fonksiyonun alanı başlangıcı
    if x % 2 = 0 then
        // If koşulu alanı
        true
    else
        // Else koşulu alanı
        False

    // Fonksiyonun alanı bitişi

// Global alanda yeni bir modül tanımı
module Modül2 =
    // Modül alanı başlangıcı
    let çiftMiTekMi x =
        // Fonksiyon alanı başlangıcı
        match x with
        // match alanı başlangıcı
        | a when a % 2 = 0 ->
            // Koşul alanı
            "Çift"
        | _ ->
            // Koşul alanı
            "Tek"
        // match alanı bitişi

        // Fonksiyon alanı bitişi

    // Modül alanı bitişi
// Müdül2 alanındaki çiftMiTekMi fonksiyonuna çağrı
Modül2.çiftMiTekMi 12

// Global alandaki çiftMiTekMi fonksiyonuna çağrı
çiftMiTekMi 12
```

F#'da modül alanı içindeki değerler ve fonksiyonlar nokta notasyonu (dot notation) ile önlerine **modül adı** yazılarak global alandan veya diğer modüller içinden kullanılabilir. Bir modül içinden ise bir üst seviyedeki (dışındaki) mo-

4

Fonksiyonel Programlama

Bu Bölümde

Desen Eşleme (Pattern Matching) 138
Kayıtlar (Record) 151
Ayrışık Bileşim (Discriminated Unıon) 157
Tiplere Davranış Ekleme 166
Tip Genelleme (Generics) 174
Sonradan Değerleme (Lazy Evaluation) 178
Neler Öğrendik? 182

Bu bölüme F#'ın en kullanışlı özelliklerinden biri olan **Desen Eşleme** yapısını inceleyerek başlıyoruz. Daha sonra ise kayıt ve bileşim tiplerini nasıl tanımlandığını ve bu tiplere nasıl davranış eklendiğini ele alıyoruz. Tipler ile ilgili önemli kavramlardan birisi olan tip genelleme (Generics) konusunu ve sekansları inceleyerek bölümü tamamlıyoruz.

DESEN EŞLEME (PATTERN MATCHING)

F#'da desenler (pattern) çok temel ve önemli bir mekanizmadır. Farkında olmasak bile örneğin `let` ile ifadelere değer atanması, fonksiyon argümanlarının işlenmesi ve `try..with` hata ayıklama yapısı gibi çok temel dil yapıları desenleri kullanır. Bu bölümde desenlerin farklı bir kullanımı olan `match..with` desen eşleme yapısını ayrıntıları ile ele alıyoruz. Desen eşleştirme yapısını bir çeşit dallanma mekanizması olarak düşünebilirsiniz. C, C++, Java ve C# ile kodlama yaptıysanız, aralarında çok ciddi farklar olsa bile, desen eşlemeyi bu dillerdeki switch..case akış kontrol yapısına benzetebilirsiniz.

Desen eşleme ifadelerinin şablonu aşağıdaki gibidir.

```
match <değer> with
| desen1 [when koşul1] -> eşlemeKodu1
| desen2 [when koşul2] ->
                         eşlemeKodu2
| desen3 [when koşul1] -> eşlemeKodu3
```

Desen eşleme ifadelerinde olası tüm desenler `match` anahtar kelimesinin hemen altında aynı hizadan başlayan | (dik tek çizgi) sembolünden sonra alt alta yazılır. Hehangi bir desen eşlendiğinde çalışması istenen kod ise `->` simgesinden sonra (eşlemeKodu1 gibi) veya hemen alt satırda (eşlemeKodu2 gibi) girintili olarak yazılır.

Aşağıdaki kod bloğunda `üsAl` isimli fonksiyon içinde `y` parametresi ile geçilen 2,3,4 değerleri ile eşleme yapılarak `x` parametresinin üssünün nasıl alındığını inceleyebilirsiniz.

```
(* 04_1_01.fsx *)

let üsAl x y =
    match y with
    | 2 -> x * x
    | 3 -> x * x * x
    | 4 -> x * x * x * x
    | _ -> x

// TEST
üsAl 2 2
üsAl 2 3
üsAl 2 4
üsAl 2 5
```

Desenlerden her biri için eğer gerek duyulursa opsiyonel olarak when ile koşul kontrolü de eklenebilir.

Aşağıdaki kod bloğunda büyükKüçükSıfırDenetimi fonksiyonu içinde sayı parametresi için desen eşleme yapılarak when koşulu ile değerin sıfıra eşit mi yoksa büyük veya küçük mü olduğu koşuluna göre işlem yapılmaktadır.

```
let büyükKüçükSıfırDenetimi sayı =
    match sayı with
    | x when x = 0 -> printfn "Değer sıfıra eşit"
    | x when x > 0 -> printfn "Değer sıfırdan büyük ve %d"  x
    | x when x < 0 -> printfn "Değer sıfırdan küçük ve %d"  x
    | _ -> printfn "Bu mükün değil ama derleyici mutlu olsun"

// TEST
büyükKüçükSıfırDenetimi 4
```

Desen eşleme ifadelerinde değerin olası desenlerden hangisi ile eşleştiğini belirlemek için match with ifadesindeki değer her bir desen için sırayla tekrar üretilip değer ve üretilen yeni değerin eşitliği karşılaştırılır.

```
open System
open System.Globalization
let metniTamSayıyaÇevir metin =
    let sonuç = Int32.TryParse(metin, NumberStyles.Any,null)
    // sonuç : bool * int

    match sonuç with
    |(true,sayı) -> Some(sayı)
    |(false,_) -> None

// TEST
metniTamSayıyaÇevir “A”
metniTamSayıyaÇevir “-42”
metniTamSayıyaÇevir “-42”
```

Yukarıdaki örneğimizde metin değerleri tam sayıya çeviren metniTamSayıyaÇevir isimli bir fonksiyon oluşturuyoruz. Bu fonkisyonun içinde standart .NET kütüphane fonksiyonu olan System.Int32.TryParse kullanarak girdi parametresindeki metni tam sayıya çevirmeye çalışıyoruz.

5

Koleksiyonlar

Bu Bölümde

Liste (List)	186
Dizi (Array)	192
Sekans (Sequence)	200
Yield! (Yield Bang)	202
Küme (Set)	204
Anahtar Değer Haritası (Map)	207
List Modülü	208
Sorgu İfadeleri (Query Expressions)	220
Neler Öğrendik?	227

Bir bakış açısına göre programlar veri işleyen ve organize eden akıllı algoritmalardan ibarettir.

Bu bölümün ilk kısmında liste, dizi, sekans, küme ve harita gibi birden fazla aynı tipten veriyi tutabildiğimiz koleksiyon tiplerini ele alıyoruz. Bölümün ikinci kısmında ise bu koleksiyonlar ile çalışan çok yetenekli ve kullanımı kolay modül fonksiyonları ile yapabileceğimiz işlemlere bakacağız.

LİSTE (LİST)

Aynı tipten elemanları barındıran tipe **liste (list)** denir. F#'da listeler sıralıdır ve içerikleri değiştirilemez.

Liste oluşturmak için aynı tipten elemanları köşeli parantezler arasında **noktalı virgül** (;) ile ayırarak yazarız. Alternatif olarak liste elemanlarının her birini yeni bir satıra yazarak noktalı virgül kullanmandan da liste oluşturulabilir.

```
(* 05_1_01.fsx *)

let liste1 = [1;2;3]

let liste2 = [
    1
    2
    3]
```

Listelerin tipleri list şeklinde yazılır. **Örneğin;** `int list` elemanlarının tipi `int` olan bir liste tipin ifade eder.

```
// elemanları int tipinden olan liste
let liste1' : int list = [1;2;3]

// elemanları string tipinden olan liste
let liste2': string list= [
    "1"
    "2"
    "3"]
```

Boş bir liste tanımlamak için liste değeri olarak `[]` kullanılır. Boş listelerin imzası `val it :'a list` şeklindedir. `'a` ibaresi listenin elemanlarının herhangi bir tipten olabileceği anlamına gelir.

```
let boşListe = []
// val boşListe : 'a list
```

> **İPUCU** F#'da tipi belli olmayan herhangi bir tipteki değerleri tanımlamak için 'a,'b,'c şeklindeki simgeler kullanılır. Tek tırnak ve bir harf şeklinde bir ifade gördüğünüzde bilin ki değerin tipi herhangi bir tip olabilir.

Elemanlarının değeri belirli bir aralıkta olan listeler **aralık operatörü** (`..`) kullanılarak tanımlanabilir.

```
(* 05_1_01.fsx *)

// Elemanları 1 ile 10 arasındaki sayılardan oluşan liste
let liste3 = [1..10]

// Elemanları 1 ile 20 arasında olan
// 2'şer artan sayılardan oluşan  liste
let liste4 = [1..2..20]

// Elemanları 1 ile 20 arasında olan
// 0.5'er artan sayılardan oluşan  liste
let liste5 = [1.0..0.5..20.0]

// Elemanları 100 ile 0 arasında
// 2'şer azalan sayılar olan liste
let liste6 = [100..-2..0]
```

DİKKAT F#'da listeler içeriği değiştirilemeyen yapılardır. Liste içeriğinde değişiklik yapan tüm fonksiyon ve operatörler yeni bir liste döndürecektir.

Listelerin elemanlarına pozisyon numaraları (indeks) kullanılarak erişilebilir. Liste elemanlarının pozisyon numaraları 0'dan başlar; 3 elemanlı bir listenin birinci elemanının pozisyonu 0, son elemanının pozisyonu da 2'dir. Listenin herhangi bir elemanına pozisyon indeksi ile erişmek için aşağıdaki şablon kullanılır.

```
liste_adı.[eleman_indeksi]
```

Aşağıdaki kod bloğunda indeksler kullanılarak liste elemanlarına nasıl erişildiğini inceleyebilirsiniz.

```
let liste = [1..10]
let birinciElaman = liste.[0]
let sonEleman = liste.[9]
let sonEleman' = liste.[10]
// Çalışma zamanı hatası!
// Liste elemanları 0'dan başlanarak indekslenir
// The index was outside the range of elements in the list.
// = operatörü karşılaştırma operatörü
```

```
// 1. elemanın değeri değiştirilemez
liste.[0] = 100

// Derleyici hatası. <- operatörü ile elemanın değeri
// değiştirilemez.
liste.[0] <- 100
```

> **NOT** Değişkenler mutable anahtar kelimesi kullanılarak tanımlanır ve değerlerini değiştirmek için <- atama operatörü kullanılır. F#'da = atama operatörü değil, mantıksal karşılaştırma operatörüdür.

```
let mutable değer = 1 // değişken tanımı

değer <- 0 // Değişken değerini değiştir
```

ELEMAN EKLEME OPERATÖRÜ ::

Bir listenin önüne listenin elemanları ile aynı tipte yeni elemanlar eklemek için **::** (cons, prepend) operatörü kullanılır. Bu operatör ile bir veya daha fazla değer listenin başına eklenir ve yeni bir liste döndürülür. **::** operatörü ile yapılan ekleme işleminin karmaşıklığı O(1)'dir, yani ekleme yapılan listenin uzunluğu ne olursa olsun yeni bir eleman ekleme performansı her zaman sabittir.

Aşağıdaki kod bloğunda **::** operatörü ile boş listelere veya içinde eleman olan listelerin başına nasıl eleman eklenebileceğini görebilirsiniz.

```
(* 05_1_01.fsx *)

// Boş liste
let liste7 = []

// 1,2,3,4 ekleniyor
let liste8 = 1::2::3::4::[]

// Elemanları 1,2,3 olan liste
let liste9 = [1;2;3]

// Listenin başına -1 ve 0 eklenir
let liste10 = -1::0::liste9
// Çıktı : val liste10 : int list = [-1; 0; 1; 2; 3]
```

:: sağdan birleşmeli bir operatördür (right-associative operator), bu nedenle listenin sonuna eleman eklemek için kullanılamaz. **::** ile sadece listenin başına yeni elemanlar eklenebilir.

7

Nesne Yönelimli Programlama

Bu Bölümde

Nesne Yönelimli Programlama Nedir? 272
Sınıf Tanımlama 272
Üye Özellikler 284
Üye Metodlar 289
Üye Erişim Kısıtlayıcıları 293
Kalıtım 296
Kontratlar/Ara Birimler 302
Sınıflar Arası Tip Dönüşümü 306
IDisposable Kullanım Desenleri 309
Neler Öğrendik? 315

Bu bölümde Nesne Yönelimli Programlama'nın (OOP) bel kemiğini oluşturan sınıfların F# ile nasıl tanımlandığını ve nesnelerin nasıl oluşturulduğunu öğreniyoruz. Sınıf ve nesnelere ilave olarak nesne yönelimli programlamanın diğer önemli kavramları olan kalıtım, kontrarlar ve tip dönüşümü yöntemleri de bu bölümde ele aldığımız diğer konulardır.

NESNE YÖNELİMLİ PROGRAMLAMA NEDİR?

Nesne Yönelimli Programlama (Object Oriented Programming - OOP) temelinde sınıflar ve bu sınıflardan üretilen nesneler vardır. Sınıflar ortak özellikleri olan nesneleri tarif etmek için kullanılan şablonlardır, nesneler ise bu şablonlar kullanılarak oluşturulur. Nesne yönelimli programlama yaklaşımında nesnelerin **davranışları** ve **özellikleri** olmak üzere iki temel kavramdan bahsederiz.

Programlama dillerinde davranışları **fonksiyonlar** ve **üye metodlar** ile özellikleri de **alanlar** ve **üye özellikler** ile modelleriz. Nesne yönelimli programlama yaklaşımında önemli bir diğer kavram da **kalıtımdır.** Kalıtım vasıtasıyla sınıflar ata olarak kabul edebilecekleri başka sınıflardan ilave davranış ve özellikleri miras alabilirler ve başka sınıflara kalıtım yoluyla kendi davranış ve özelliklerini iletebililer. Sınıflar diğer sınıflara yerine getirmeyi taahhüt ettikleri davranışları ve sundukları özellikleri **interface** adı verilen özel kontrat tipleri ile bildirebilirler.

Sınıflar nesneleri modeller, bu nedenle sınıflar üzerinden tanımlanan özellik ve metodları kullanmak için çoğunlukla ilgili sınıftan bir nesneye yani **sınıf örneğine** (instance) ihtiyaç vardır. Sınıf örneği üzerinden kullanılan bu özellik ve metodlara **örnek özellikleri** veya **örnek metodları** denir. Ancak, bazı durumlarda sınıf örneği olmadan da sınıfın kendisi için özellik ve metod tanımlamak gerekebilir. Bu tür özellik ve metodlara da **statik özellikler** veya **statik metodlar** denir.

Bu bölümde sınıfları merkez alarak F#'ın sunduğu temel nesne yönelimli programlama yapılarını ele alıyoruz.

SINIF TANIMLAMA

Sınıflar diğer F# tipleri gibi type anahtar kelimesi kullanılarak aşağıdaki söz dizimi şablonuna göre tanımlanır.

```
// Sınıf tanımı:
type [access-modifier] type-name [type-params] [access-modifier] ( parameter-list ) [ as identifier ] =
[ class ]
[ inherit base-type-name(base-constructor-args) ]
[ let-bindings ]
[ do-bindings ]
member-list
...
[ end ]
```

```
// karşılıklı öz yinelemeli sınıf tanımları:
type [access-modifier] type-name1 ...
and [access-modifier] type-name2 ...
...
```

Varsayılan Kurucu Fonksiyonlar

Sınıflardan nesne üretmek için kullanılan özel foksiyonlara **constructor** adı verilir. Biz bu fonksiyonları **kurucu fonksiyonlar** olarak adlandıracağız. F#'da sınıflar için mutlaka ama mutlaka en az bir tane kurucu fonksiyon tanımı yapılmalıdır. Kurucu fonksiyonların görevi nesneyi dışarıdan verilen parametre değerlerine sahip özellikler ile ve gerekiyorsa bir takım ilklendirme işlemleri yapıldıktan sonra üretmektir.

Aşağıdaki örneğimizde `Dikdörtgen` isimli bir sınıf tanımlıyoruz.

```
(* 07_1_01.fsx *)

type Dikdörtgen (x:int,y:int) =
    ...
    ...
```

Sınıf isminden hemen sonra gelen `(x:int,y:int)` ifadesi ile `Dikdörtgen` sınıfı için `x` ve `y` isimli parametreleri olan dolaylı bir kurucu fonksiyon tanımı yapılır. Bu kurucu fonksiyon aynı zamanda sınıfın varsayılan kurucu fonksiyonudur. Kurucu fonksiyonun `x` ve `y` isimli parametrelerine sınıf içinde normal birer değer gibi doğrudan erişilebilir.

```
type Dikdörtgen(x:int,y:int) =
    // sınıfın X ve Y isimli üye özellikler
    member this.X = x
    member this.Y = y
```

Sınıf tipinden bir nesne oluşturmak için sınıf adı ve kurucu fonksiyon parametre değerleri aşağıdaki gibi tanımlanmalıdır. Nesne oluştururken `new` anahtar kelimesinin kullanımı opsiyoneldir.

```
let diktörgen = new Dikdörtgen(2,3)

let diktörgen' = Dikdörtgen(2,3)
```

8

Gelişmiş Fonksiyonel Programlama Yöntemleri

Bu Bölümde

Aktif Desenler 318
Kuyruk Özyenilemeli Fonksiyonlar 325
Aktarım Operatörleri 327
Fonksiyon Kompozisyonu 332
Monadlar ve Hesaplama İfadeleri 334
İpuçları 347
Neler Öğrendik? 357

Bu bölümde günlük kodlama pratiklerimizi iyileştirip yazılan kodu hem daha şık hale getiren hem de kod miktarını önemli ölçüde azaltarak programlarımıza esneklik kazandıran ileri seviye fonksiyonel programlama kavram ve yöntemlerini ele alıyoruz.

Aktif desenler, aktarım operatörleri ve kompozisyon operatörleri yazdığınız koda bakış açınızı kökünden değiştirme potansiyeline sahip oldukları için bu bölümün ön plana çıkan konularıdır.

Aktif Desenler

F#'ın `match...with` ile sunduğu desen eşleme yapısı F# kullanıcı ve geliştirici topluluğu tarafından dilin en önemli özelliklerinden birisi olarak kabul edilir ve anlatılır. F# topluluğunun bu konuda kesinlikle haklı olduğunu kabul etmeliyiz, çünkü desen eşleme sayesinde basit ve temel F# tiplerininin değerlerini, tanımlanan desenlere göre daha önce de değindiğimiz derleyici destekli ve akıllı algoritma ile sökerek koşullu dallanma yapıların kodlayabiliyoruz. Ayrıca, `match...with` ile desen eşleme genelde prosedürel dillerde bulunan **if/else** koşullu dallanma yapısına daha esnek ve daha fonksiyonel bir alternatif olarak da ön plana çıkıyor.

Ancak, `match...with` varsayılan hali ile sadece temel ve basit tipler ile sınırlı olduğu için bazı durumlarda yetersiz kalabiliyor. Gelin şimdi bu yapının yetersiz kaldığı bir örnek kurgulayıp daha sonra da aktif desenler ile bu kısıtlamalardan nasıl kurtulabileceğimizi görelim.

Problemimiz metin tipinden bir değerin tam sayı (int) mı yoksa true/false mantıksal (bool) bir değer mi olduğunu tespit edip değeri ekrana yazdırmak. Bunun için aşağıdaki gibi `SayıMıBoolMu` isimli bir fonksiyon oluşturuyoruz. Bu fonksiyon sırası ile her iki tipe dönüşümü `TryParse` metodlarını kullanarak deniyor, daha sonra da iki denemenin sonuçlarını bir değer grubunda birleştiriyor. Son olarak da birleştirilen sonucu kullanarak desen eşleştirme yapılıyor ve desenlere göre girdi değerinin sayı mı yoksa mantıksal bir değer mi olduğu belirleniyor.

Bu fonksiyondaki göze batan en önemli sorun girdinin tipine ve değerine karar veren kodun desen eşleme yapan kod olmaması. Desen eşleme ancak değerler ayrı ayrı belirlenip birleştirildikten sonra yapılabiliyor.

```
(* 08_1_01.fsx *)

open System
open System.Globalization

let SayıMıBoolMu (girdi:string) =
    let (sayıMı,değer1) = Int32.TryParse(girdi,NumberStyles.Any,null)
    let (boolMu,değer2) = Boolean.TryParse(girdi)

    let sayıDeğeri = if sayıMı then Some(değer1) else None
    let boolDeğer = if boolMu then Some(değer2) else None
    let sonuç = (sayıMı,boolMu,sayıDeğeri,boolDeğer )
```

```
    match sonuç with
    | (true,_,Some(değer),_) -> printfn "int, değer = %d" değer
    | (_,true,_,Some(değer)) -> printfn "bool, değer = %A" değer
    | _ -> printfn "Sonuç int veya bool değil!"

// TEST

SayıMıBoolMu "12"
SayıMıBoolMu "true"
SayıMıBoolMu "Mahmut Tuncer"
```

SayıMıBoolMu fonksiyonunu daha sade bir şekilde aşağıdaki gibi kodlamak istediğimizde Int32.TryParse veya Boolean.TryParse fonksiyonlarının desen olarak kullanılamayacağını keşfediyoruz, çünkü derleyici "The field, constructor or member 'TryParse' is not defined." şeklinde bir hata veriyor.

```
(* 08_1_01.fsx *)

open System
open System.Globalization

let SayıMıBoolMu (girdi:string) =
    match girdi with
    | Int32.TryParse(girdi,NumberStyles.Any,null) as sonuç
        when fst sonuç -> printfn "int, değer = %d" snd sonuç
    | Boolean.TryParse(girdi) as sonuç
        when fst sonuç-> printfn "bool, değer = %A" snd sonuç
    | _ -> printfn "Sonuç int veya bool değil!"
```

Desen eşleme konusunu ele aldığımız bölümde desenden değer sökme algoritmasının nasıl çalıştığını hatırlarsanız "The field, constructor or member 'TryParse' is not defined." hatasının tam olarak ne anlama geldiğini daha iyi kavrayabilirsiniz. Bu hata bize özetle şunu anlatıyor; TryParse fonksiyonu kullanılarak verilen girdiler ile bir değer üretilip karşılaştırma yani desen eşleme yapılamıyor.

Bu örnek ile varsayılan desen eşleme yönteminin hangi koşullarda yetersiz kaldığını gördüğümüze göre hedefimizi aşağıdaki gibi bir fonksiyon oluşturmak olarak belirleyebiliriz. SayıMıBoolMu fonksiyonu ilave dönüşüm ve birleştirmeler gerektirmeden Int ve Bool gibi iki desene göre girdinin tipini ve değerini dönüştürebilmeli.

9

Asenkron ve Paralel Programlama

Bu Bölümde

Giriş ve Temel Kavramlar	360
Thread Sınıfı	363
Asenkron İş Akışları	379
Task Sınıfı ve Paralel Programlama	394
MaılboxProcessor Sınıfı	401
Neler Öğrendik?	409
Terimler	410

Bu bölümde, arka planda asenkron işlem yapmak için F# ve .NET'in sunduğu Thread sınıfı, asenkron ifadeler, arka plan görevleri, paralel işlem kütüphanesi ve MailboxProcessor sınıfı konularını ele alıyoruz.

Bölüm konuları eş zamanlı işlem oluşturmak için kullanılan en alt seviyedeki Thread sınıfı ile başlayıp kademeli bir yöntem takip edilerek MailboxProcessor sınıfının anlatımı ve örnekleri ile tamamlanacaktır.

GİRİŞ VE TEMEL KAVRAMLAR

Asenkron programlama yaklaşımı kullanılan dilden bağımsız olarak tüm programcıların hem yazmakta hem de yazılan kodu anlamakta en çok zorlandığı kavramların başında gelir. Bu korku programcılık kültürü açısından temelleri olan ve öğretilmiş bir korkudur. Asenkron programlamanın zorluğu hem kullanıcılar hem de programcılar için oldukça ciddi sonuçlar doğurma potansiyeli olan aşağıdaki iki soruna indirgenebilir.

» Birden fazla büyüklük tarafından paylaşılan ortak verinin öngörülemez biçimde değişmesi,

» Deadlock adı verilen ve iki büyüklüğün karşılıklı olarak sonsuza kadar birbirini beklemesine neden olan kilitlenme,

Programcılık mesleği ve kültürünün ötesinde klasik eğitim yöntemleri problemleri izole etmeyi ve sonra da ardışıl adımlardan oluşan çözümler üretmeyi öğretiyor. Bu nedenle, genelde problemleri daha küçük problemlere bölüp çözümlerimizi de birbirini takip eden adımlardan oluşturmayı tercih ediyoruz.

Doğal olarak genel eğilimimizin aksine asenkron veya paralel çalışan mekanizmalar kurguladığımızda normalden daha fazla orandaki hatalar ve ona eşlik eden korku peşimizi bırakmıyor. Bu genel durumdan daha özelde programcılık veya yazılım mühendisliği müfredatlarına baktığımızda ise yazılım geliştirmenin temeli olarak hala asenkron çalışma ve paralel işlem kabiliyeti prensiplerine uygun olmayan, göreceli eski ve klasik yaklaşımı yansıtan algoritma ve veri yapılarının öğretildiğini söyleyebiliriz.

Ancak, devir artık büyük veri, daha fazla işlem gücü ve doğal olarak asenkron çalışma ve paralel işlem yapabilme devri. Bu nedenle öğretilmiş korkularımızı terk edip kullandığımız programlama dillerinin bize sağladıkları kullanımı kolay ve esnek asenkron yapıları öğrenip programlarımıza paralel işlem yapabilme kabiliyeti kazındırmalıyız.

Bilgisayar olarak adlandırabileceğimiz tüm cihazların kalbi olan ve işlem gücünü temsil eden **Merkezi İşlem Birimleri** (CPU) bundan 15-20 yıl öncesine kadar bir anda tek bir işlem yapabilme kabiliyetine sahiptiler. Ancak, son dönemde malzeme bilimindeki gelişmeler ile birlikte CPU dediğimiz birim içinde birden fazla çekirdek barındıran daha karmaşık bir yapı haline geldi. Çok çekirdekli (multi core) CPU'lar sayesinde artık gerçek anlamda aynı anda birden fazla işlem yapmak mümkün. Çok çekirdekli bir cihazı içinde birden fazla bilgisayar barındıran ve paralel işlem yapma kabiliyeti olan süper bir bilgisayar olarak adlandırabiliriz.

F#'da asenkron ve paralel programlama yapmak için kullanabileceğiniz yapıları en alt seviyedekinden en üst seviyedekine göre şöyle özetleyebiliriz.

» Thread'ler (iş parçacıkları)

» Async ifadeler

» Paralel programlama kütüphanesi

Gelin şimdi bu yöntemlerin ayrıntılarını ele alıp, varsa, asenkron programlama ile ilgili korkularımızdan kurtulalım.

Thread Sınıfı

Thread'ler (iş parçacıkları) daha önce de değindiğimiz gibi programlarımıza asenkron işlem kabiliyeti kazandırmak için kullanabileceğimiz en alt seviyedeki yapılardır. Thread'lerin çalışma süreleri, çalışma sıraları ve kullanacakları kaynaklar doğrudan işletim sistemi tarafından yönetilir. F#'da thread'lerin ile çalışmak için .NET'in **System.Threading** alan adı içinde sunduğu `Thread` sınıfını ve bu sınıfın fonksiyonlarını kullanmalıyız.

Yeni bir thread oluşturumak için `Thread` sınıfından bir nesne üretip kurucu fonksiyonuna parametre olarak arka plan işlemini yapacak fonksiyonu veriyoruz. Oluşturduğumuz thread nesnesi `Start` ile başlatıldığında kurucu fonksiyona parametre olarak geçilen fonksiyon çalıştırılır.

Aşağıdaki örneğimizde asenkron olarak arka planda yaptığımız işlem 1 ile 5 arasında sayıları sırası ile ekrana yazdırmak. Bu işlemi yapmak için `birdenBeşeKadarSay` isimli `unit -> unit` imzalı bir fonksiyon oluşturuyoruz. İkinci fonksiyonumuz olan `threadOluştur` ile `Thread` tipinden bir nesne oluşturup `Thread` sınıfı kurucusuna `birdenBeşeKadarSay` fonksiyonunu parametre olarak geçiyoruz. `Thread` nesnesini oluşurduktan sonra da `Thread.Start` `statik` metodu ile `threadi` ve dolayısıyla arka plan işini başlatıyoruz.

```
(* 09_2_01.fsx *)

open System.Threading

let birdenBeşeKadarSay() =
    for i in [1..5] do
        Thread.Sleep(100)
        let threadId = Thread.CurrentThread.ManagedThreadId
        printfn "Thread %d : %d" threadId i
```

DİKKAT Her problem paralelleştirmeye uygun olmayabilir. **Örneğin;** aynı dosyaya erişimi gerektiren işlemlerde ortak değişken/kaynak kullanımı zorunludur ve paralelleştirme mümkün olmayacaktır.

KARŞILIKLI KİLİTLEME

Ortak değişken ve kaynaklara erişimin kontrol altına alınması önceki başlıkta ele aldığımız dil yapıları ile oldukça kolaydır. Ancak, erişimin senkronize edilmesi gizli bir tehlikeyi de barındırır; karşılıklı kilitlenme (deadlock). Deadlock durumu iki thread'in karşılıklı olarak ortak kaynağa erişim için birbirlerini sonsuza kadar beklemeye başladıklarında ortaya çıkar.

Aşağıdaki örneğimizde a ve b ortak erişilen değişkenlerdir. İlk thread'in çalıştıracağı `işlemYap` içinde önce a kilitlenir sonra da b kilitlenir. İkinci thread'in çalıştıracağı `işlemYap'` içinde ise önce b kilitlenir sonra da a kilitlenir. Karşılıklı kilit durumunu oluşturmak için `işlemYap` içinde a kilitlendikten sonra 3 saniye thread'i bekletiyoruz. İlk thread'i `QueueUserWorkItem` ile kuyruğa ekledikten sonra a'nın kilitlenmesine zaman tanımak için 1 saniye bekleyip ikinci thread'i çalıştırıyoruz. Birinci thread içindeki 3 saniyelik süre sonunda b için kilit alınmaya çalışılır ancak ikinci thread b'yi kilitlediği için birinci thread bu noktada beklemeye başlar, bu sırada ikinci thread de a zaten birinci thread tarafından kilitlendiği için beklemeye başlar. Sonuç olarak thread'ler karşılıklı olarak birbirini beklemeye başlar.

```
(* 09_2_12.fsx *)

open System.Threading

let a = ref 0
let b = ref 0

let işlemYap(v:obj) =
    printfn "Thread 1: a'yi kilitle"
    lock (a) (
                fun () ->
                    printfn "Thread 1: a kilitli"
                    Thread.Sleep(3000) |> ignore

                    printfn "Thread 1: b'yi kilitle"
                    lock (b) (
                                fun () ->
                                    printfn "Thread 1: b kilitli"
```

```
                                    a := !a + (v :?> int)
                                    b := !a + !b + (v :?> int)
                                )
            )
let işlemYap'(v:obj) =
    printfn "Thread 2: b'yi kilitle"
    lock (b) (
                fun () ->
                    printfn "Thread 2: b kilitli"
                    Thread.Sleep(3000) |> ignore

                    printfn "Thread 2: a'yi kilitle"
                    lock (a) (
                                fun () ->
                                    printfn "Thread 2: a kilitli"
                                    a := !a + (v :?> int)
                                    b := !a + !b + (v :?> int)
                                )
            )

ThreadPool.QueueUserWorkItem(WaitCallback işlemYap,5)
Thread.Sleep(1000)
ThreadPool.QueueUserWorkItem(WaitCallback işlemYap',5)
()
```

Deadlock durumlarının oluşma ihtimali sizi ürkütmesin zira işlem sıralamasını değiştirerek ve thread'lerin ortak eriştikleri kaynaklar iyi incelenip mümkünse minimize edilerek deadlock oluşma ihtimali ortadan kaldırılabilir.

Asenkron İş Akışları

F#'ın asenkron programlama modeli önceki bölümlerde bahsettiğimiz `Thread` sınıfı ve diğer yapıların üstüne inşa edilen `Async` modülü içindeki fonksiyonlar ve `async` ifadelerden (async expression) oluşur. F#'da asenkron iş akışlarının temelinde `Async<'T>` sınıfı yer alır. Bu sınıf arka planda çalıştırılması istenen işlemleri temsil eder. `Async<'T>` olarak oluşturulan arka plan işlemleri oluşturuldukları anda otomatik olarak çalıştırılmazlar, bunun yerine `Async.RunSynchronously` benzeri başlatıcı fonksiyonlar kullanılarak çalışmaları sağlanır. `Async<'T>`'de `'T` arka planda çalışan işlemin sonucu olan dönüş değerinin tipini ifade eder. `'T` herhangi bir tip olabileceği gibi `unit` de olabilir.

Async İfadeler

Aşağıda WebClient sınıfını kullanarak adresi verilen web sayfasının içeriğini indiren örnek indir fonksiyonunu görebilirsiniz. Bu hali ile fonksiyonun kendi başına asenkron işlem yapma kabiliyeti yoktur ve eş zamanlı işlemlerde ancak ve ancak Thread sınıfından nesneler ile kullanılabilir.

```
(* 09_3_01.fsx *)
open System
open System.Net

let indir adres =
    use istemci = new WebClient()
    let uri = Uri(adres)
    istemci.DownloadString(uri)
```

Thread sınıfından nesnelere gerek kalmadan yukarıdaki fonksiyonu F#'ın sunduğu async{...} ifadesi ile asenkron hale getirebilirz.

```
(* 09_3_02.fsx *)
open System
open System.Net

let asyncİndir adres =
    async{
        use istemci = new WebClient()
        let uri = Uri(adres)
        let! html = istemci.AsyncDownloadString(uri)
        return html
    }
```

Asenkron ifadelerin içinde let, use ve do ifadelerine ilave olarak aşağıdaki dil yapılarını da kullanabiliriz.

» **let!**, farklı bir asenkron bağlamda çalışan işlemin sonucunu bir ifadeye değer olarak bağlar. Asenkron işlemlerin sonucu let ile tanımlı bir ifadeye bağlanırsa değerin tipi Async<'T> olur ve program akışı asenkron ifadenin tamamlanmasını beklemez. Ancak, let! kullanarak program akışının asenkron işlem tamamlanana kadar bekletilmesini ve işlem bitince de Async<'T> tipi tarafından çevrelenen 'T tipinden değerin ifadeye bağlanması sağlanır.