2017

F# İLE FONKSİYONEL PROGRAMLAMA

ALİ ÖZGÜR

GİZLİLİK VE KULLANIM SÖZLEŞMESİ

“F# ile Fonksiyonel Programlama” kitabının bu kopyası ön izleme amaçlı olarak sadece alıcısına özel bir dağıtım olarak paylaşılmıştır. Bu kopyanın tamamını veya bir kısmını yazarından yazılı izin almadan kullanmak, alıntılamak, elektronik ortamda veya basılı olarak çoğaltıp dağıtmak veya satmak yasaktır.

Bu dokümanı elektronik ortamda veya basılı olarak teslim alan kişi bu gizlilik sözleşmesini kabul etmiş sayılır.

İÇİNDEKİLER – Ön İzleme Versiyonu

- 1.Bölüm : Giriş
  - 1.1 F# ile Tanışma
  - 1.2 Kısa F# Tarihçesi
  - 1.3 Neden F#?
  - 1.4 Fonksiyonlara Matematiksel Bakış
  - 1.5 Fonksiyonların İlginç Özellikleri
  - 1.6 Fonksiyonel Programlama Nedir?
- 2.Bölüm : F# Geliştirme Platformu
  - 2.1 Derleyici ve Yorumlayıcı Kavramları
  - 2.2 FSC - F# Derleyicisi
  - 2.3 FSI - F# Etkileşimli Ortamı
  - 2.4 F# Standard Dosya Uzantıları
  - 2.5 Derleyici ve Etkileşimli Ortam Değişkenleri
  - 2.6 Geliştirme Araçları
  - 2.7 Merhaba F#
- 3.Bölüm : F# Temelleri
  - 3.1 Söz dizimi kuralları
  - 3.2 Basit Veri Tipleri
  - 3.3 Fonksiyonlar
  - 3.4 Fonksiyonların İleri Seviye Kullanımı
  - 3.5 Temel Veri Tipleri
  - 3.6 Yapısal Eşitlik
  - 3.7 Kod Organizasyonu

İÇİNDEKİLER – Tam Versiyon

- 4.Bölüm : Fonksiyonel Programlama
  - 4.1 Desen Eşleme (Pattern Matching)
  - 4.2 Kayıtlar (Record)
  - 4.3 Ayrışık Bileşim (Discriminated Union)
  - 4.4 Tiplere Davranış Ekleme
  - 4.5 Tip Genelleme (Generics)
  - 4.6 Sonradan Değerleme (Lazy Evaluation)
  - 4.7 Sekanslar (Sequences)
  - 4.8 yield! (yield bang)
- 5.Bölüm : Koleksiyonlar
  - 5.1 Liste (List)
  - 5.2 Dizi (Array)
  - 5.3 Sekans (Sequence)
  - 5.4 yield! (yield bang)
  - 5.5 Küme (Set)
  - 5.6 Anahtar Değer Haritası (Map)
  - 5.7 List Modülü
  - 5.8 Sorgu İfadeleri (Query Expressions)
- 6.Bölüm : Genel Amaçlı Programlama
  - 6.1 Değişkenlere Gerçekten İhtiyacımız Var Mı?
  - 6.2 Değer Tipleri ve Referans Tipleri
  - 6.3 Değişkenler
  - 6.4 .NET Koleksiyonları
  - 6.5 Döngü Yapıları (For ve While)
  - 6.6 Koşullu Dallanma Yapıları (If/Else)
  - 6.7 İstisnalar (Exceptions)
  - 6.8 Ölçü Birimleri
- 7.Bölüm : Nesne Yönelimli Programlama
  - 7.1 Nesne Yönelimli Programlama Nedir?
  - 7.2 Sınıf Tanımlama
  - 7.3 Üye Özellikler
  - 7.4 Üye Metodlar
  - 7.5 Üye Erişim Kısıtlayıcıları
  - 7.6 Kalıtım
  - 7.7 Kontralar/Ara Birimler
  - 7.8 Sınıflar Arası Tip Dönüşümü
  - 7.9 IDisposable Kullanım Desenleri
- 8.Bölüm : Gelişmiş Fonksiyonel Programlama Yöntemleri
  - 8.1 Aktif Desenler
  - 8.2 Kuyruk Özyenilemeli Fonksiyonlar
  - 8.3 Aktarım Operatörleri

  - 8.4 Fonksiyon Kompozisyonu
  - 8.5 İpuçları
- 9.Bölüm : Asenkron ve Paralel Programlama
  - 9.1 Giriş ve Temel Kavramlar
  - 9.2 Thread Sınıfı
  - 9.3 Asenkron İş Akışları
  - 9.4 Task Sınıfı ve Paralel Programlama
  - 9.5 MailboxProcessor Sınıfı

```
Kodlar    : https://github.com/aliozgur/fsharp_kitap
Ön izleme : http://aliozgur.net/fsharp_kitap/
```

İletişi : aliozgur.net | Github | Twitter | Mail

1. Bölüm: Giriş

Bu bölümde önce F#'ın kısa tarihçesine göz atıp, "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.

1.1 F# ile Tanışma

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 birşeyler yazdırmak işte bu kadar kısa ve basit. Gelin şimdi F# ile günlük çalışmamızda sık sık karşılaşacağımız yapılara göz atıp F#'ın dil yapıları ile tanışalım.

```
(* 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"
```

```
// ======== Listeler ============
// Köşeli parantez ile liste tanımlanır
// liste elemanlarını ; ile ayrılır.
let pozitifSayılar = [1;2;3;4;5]

//Elemanları 1 ile 100 arasındaki sayılar olan liste.
let liste100 = [1..100]

// Elamanları 1 ile 100 arasında olan ve 1'den itibaren
// 2 artarak oluşturulan sayılar olan liste.
let liste101 = [1..2..100]

// :: operatörü var olan listenin başına
// 0 değerini ekleyerek yeni bir liste oluşturur.
// doğalSayılar listesinin içeriği [0;1;2;3;4;5] olur.
let doğalSayılar = 0 :: pozitifSayılar

// @ operatörü ile iki liste birleştirilip
// yeni bir liste oluşturulur.
// tamSayılar listesinin içeriği [-5;-4;-3;-2;-1;0;1;2;3;4;5]
// olur.
let tamSayılar = [-5;-4;-3;-2;-1] @ doğalSayılar

// DİKKAT: liste ve dizilerin elemanlarını tanımlarken
// virgül yerine noktalu virgül kullanılır.

// ======== Fonksiyonlar ========
// "let" anahtar kelimesi ile değer ifadelerinin yanı sıra
// ismi olan fonksiyonlar da tanımlanır.

// Fonksiyon tanımında parantez, süslü parantez veya
// noktalı virgül kullanılmaz.
let küp x = x * x * x

 // Fonksiyonu çalıştıralım.
 // Fonksiyona parametre geçerken parantez kullanmıyoruz!
küp 3

// ekle fonksiyonunu çağırırken parametreleri geçmek için
// parantez kullanılmaz.
// (1,2) 1 ve 2 değerlerini girdi olarak kullanmak anlamına gelmez
// (1,2) şeklindeki ifade ile değer grubu (tuple) tanımlanır.
let ekle x y = x + y
ekle 2 3
```

```
// Birden fazla satıra yayılmış bir fonksiyon tanımlamak
// için girintiler (indent) kullanılır.
// Kod satırlarının bitişini belirtmek için ; kullanılmaz.
let çiftSayılar liste =
    // çiftMi fonksiyonunu iç fonksiyon olarak tanımla.
    let çiftMi x = x%2 = 0

    // filter fonksiyonu List modülü içinde tanımlıdır.
    // filter girdi olarak bir fonksiyon parametresi ve
    // bu fonksiyonu uygulayacağı listeyi alır.
    List.filter çiftMi liste

// Fonksiyonu çalıştır.
çiftSayılar pozitifSayılar


// Parantezleri işlem önceliğini belirtmek için kullanılabilir.
// Parentezli ifadelerde öncelik içten dışa ve sağdan sola hesaplanır.
// Önce en içte ve sağdaki parantezli ifade çalıştırılır.

// Aşağıdaki örnekte önce List.map işleminin yapılması
// sonra da List.sum işleminin yapılması isteniyor.
// Parantezler kullanılmazsa "List.map" fonksiyonu
// "List.sum" fonksiyonuna ilk parametre olarak geçilmiş olur.
let küplerinToplamı =
   List.sum ( List.map küp [1..100] )

// Bir fonksiyonun çıktısını sonraki fonksiyona
// "|>" (ileri aktarım) operatörü ile aktarılır.
// Küplerin toplamı fonksiyonu |> kullanılarak
// aşağıdaki gibi de yazılabilir


// [1..100] listesini List.map fonksiyonuna
// ikinci parametresi olarak aktar.
// List.map fonksyonunun birinci parametresi küp fonksiyonudur.
// List.map sonucunu List.sum fonksiyonuna girdi olarak aktar.
let küplerinToplamı2 =
   [1..100] |> List.map küp |> List.sum

// "fun" anahtar kelimesini kullanılarak
// adsız (anonim) fonksiyonlar tanımlanır.
let küplerinToplamı3 =
    // fun x -> x * x * x anonim bir fonksiyon tanımıdır.
   [1..100] |> List.map (fun x->x*x*x) |> List.sum
```

```
// Fonksiyonların içinde yerel fonksiyonlar tanımlanabilir.
let birArttır x =
    // Yerel fonksiyon tanımı.
    // Kabuk fonksiyon olan birArttır'ın parametrelerine erişebilir.
    let birArttırİçFonksiyon() =
        x + 1

    // Yerel fonksiyonu kabuk fonksiyon içinden kullan.
    birArttırİçFonksiyon()

// Fonksiyonu çağır.
birArttır 2


// Öz yinelemeli fonksiyon tanımlamak için
// "rec" anahtar kelimesi kullanılır.
// Aşağıdaki fonksiyon öz yinelemeli olarak
// faktöriyel hesabı yapar.
let rec fact x =
    if x <= 1 then 1 else x * fact (x - 1)

// F#'da fonksiyonların dönüş değerleri dolayılı olarak
// belirlenir, bu nedenle değer döndürmek için "return"
// benzeri bir anahtar kelimeye kullanılmaz.
// Bir fonksiyon bloğundaki son ifade her zaman dönüş değerini oluşturur.

// ======== Desen Eşleme (Pattern Matching) ========
// Desen eşleme için Match..with.. yapısı kullanılır.
let basitDesenEşleme =
   let x = 1
   match x with
    | 1 -> printfn "x'in değer 1"
    | 2 -> printfn "x'in değeri 2"
    // _ simgesi herhangi bir değeri eşlemek için
    // yer tutucu olarak kullanılır
    | _ -> printfn "x'in değeri 1 veya 2 değil"

// Some(<değer>) ve None, C benzeri dillerdeki null veya
// Pascal benzeri dillerdeki nil değeri gibi düşünülebilir.
// F#'da Some/None dil yapısına option (opsiyon) denir.
let geçerliDeğer = Some(42)
let geçersizDeğer = None

// Aşağıdaki örnekte Some ve None desen eşleme ile
// birlikte kullanılmaktadır.
// Desen eşleme yaparken Some ifadesinin çevrelediği değeri
// kolayca söküp kullanabiliriz
```

```
let optionKullanarakEşleme girdi =
   match girdi with
    | Some i -> printfn "Girdi değeri = %d" i
    | None -> printfn "Girdi değer belirtilmemiş"

// Ekrana "Girdi değeri = 42" basılacak.
optionKullanarakEşleme geçerliDeğer

// Ekrana "Girdi değer belirtilmemiş" basılacak.
optionKullanarakEşleme geçersizDeğer

// ========= Karmaşık Veri Tipleri  =========

// Değer grupları (tuple) farklı tiplerde birden fazla
// değer barındırabilir.
// Değer grubu tanımlanırken virgül kullanılır.
let ikili = 1,2
let üçlü = "a",2,true

// Değer grupları tanımlarken parantez kullanımı opsiyoneldir
let dörtlü = ("a",2,true,System.DateTime.Now)

// Kayıt tiplerinin (record) alanları vardır.
// Alanları birbirinden ayrımak için noktalı virgül kullanılır.
type Öğrenci = {Ad:string; Soyad:string; Numara:int}

let öğrenci1 = {Ad="Arda"; Soyad="Özgür";Numara=124}

// Bileşimler (union) birden fazla seçenek tanımlanabilmesini sağlar.
// Bunlara ayrışımlı bileşimler (discriminated union) de denir.
// Bileşimlerin seçenkleri dikine çizgi (|) simgesi ile birbirinden
// ayrıştırılırlar.
type Derece =
    | C of float
    | F of float
let dereceSantigrad = C 20.0
let dereceFahrenheit = F 68.0

type Kişi = {Ad:string;Soyad:string}
```

```
// Tipler öz yinelemeli olarak karmaşık yapılar (örneğin ağaç yapısı)
// oluşturacak şekilde tanımlanabilir.
// Aşağıdaki örnekte İşçi ve Yönetici olarak ayrışan
// öz yinelemeli bir bileşim tanımlanmıştır.
// Yönetici değer olarak Çalışan listesi alabilir.

type Çalışan =
  | İşçi of Kişi
  | Yönetici of Çalışan list

let kişi = {Kişi.Ad="Ali";Soyad="Özgür"}
let işçi = İşçi kişi

// ========= Ekrana çıktı gönderme =========
// F# standard kütüphanesindeki printf/printfn fonksiyonları
// ekrana metin yazedırmak için kullanılır
printfn "Bir int %i, bir float %f ve bir bool %b" 42 3.14 true
printfn "Metin %s ve tipi ile ilgilinemiyorum : %A" "Merhaba Dünya"
[1;2;3;4;5]

// F# tüm karmaşık tipleri ekrana düzgün formatlayarak yazdırır
printfn "ikili=%A,\nkişi=%A,\nişçi=%A"  ikili kişi işçi

// Formatlanmış metni çıktı olarak döndürürmek için
// F# standard kütüphanesindeki sprintf fonksiyonu kullanılabilir
let çıktı1 = sprintf "Bir int %i, bir float %f ve bir bool %b" 42 3.14
true
let çıktı2 = sprintf "Metin %s ve tipi ile ilgilinemiyorum : %A" "Merhaba
Dünya" [1;2;3;4;5]
let çıktı3 = sprintf "ikili=%A,\nkişi=%A,\nişçi=%A"  ikili kişi işçi
```

Yukarıdaki örneklere bakıldığında yapılan işin karmaşıklığı ve ayrıntısı artsa bile F#'ın sadeliğinden ödün vermediğini ve kodun oldukça şık göründüğünü söyleyebiliriz. Yazılan kodun şıklığı, estetik görünüme ilave olarak kolay okunan, kolay anlaşılan ve kolay yönetilebilen kodu simgeler.

printfn Fonksiyonu

Kitaptaki kod örneklerinde yoğun olarak F# standard kütüphanesinde yer alan **printfn** fonksiyonunu kullanıyoruz. Bu fonksiyon konsol (terminal) ekranına metin yazdırmak için kullanılır. **printfn**'in varyantı olan **printf** ve **sprintf** fonksiyonlarını da örneklerimizde yoğun olarak kullanacağız.

- **printf**, terminale formatlanmış bir metin yazdırmak için kullanılır. Parametre olarak format metni ve format metnindeki yer tutucuların değerlerini alır.
- **printfn**, terminale formatlanmış bir metin yazdırıp imleci bir sonraki satıra konumlamak için kullanılır. Parametre olarak format metni ve format metnindeki yer tutucuların değerlerini alır.
- **sprintf**,parametre olarak verilen format metnini format metnindeki yer tutucuların değerleri ile yorumlayarak yeni bir metin üretir.

```
printfn "F# ile fonksiyonel programlama"
printfn "Ali Özgür"
// --- ÇIKTI ---
(*
F# ile fonksiyonel programlama
Ali Özgür
*)

printf "F# ile fonksiyonel programlama"
printf ", Ali Özgür"
// --- ÇIKTI ---
//F# ile fonksiyonel programlama, Ali Özgür

let yazar = "Ali Özgür"
let kitapBilgisi = sprintf "F# ile fonksiyonel programlama, %s" yazar

// kitapBilgisi ifadesinin değeri
//"F# ile fonksiyonel programlama, Ali Özgür"
```

printf, **printfn** ve **sprintf** fonksiyonlarında format metni içindeki yer tutucular (%d, %s, %A gibi) ile bu yer tutucuların değer parametrelerinin tip ve adet uyumluluğu derleme anında kontrol edilir. Format metni ile değerler arasında uyumsuzluk varsa derleyici hata verir.

Aşağıdaki tabloda format metni içinde kullanılabilen tüm yer tutucuların listesi verilmiştir.

Yer tutucu	Açıklama	Örnek	Sonuç
%d, %i	Tam sayı değerleri için kullanılır	printfn "Sayı değeri %d" 5	Sayı değeri 5
%f	Ondalık basamaklı sayılar için kullanılır	printf "Sayı = %f" 3.14	Sayı = 3.140000
%x	Tam sayı değerlerini 16'lık tabanda ekrana basar	printf "255 = %x" 255	255 = ff
%o	Tam sayı değerlerini 8'lık tabanda ekrana basar	printf "255 = %0" 255	255 = 377
%s	Metinler için kullanılır	printf "Metin = %s" "AB"	Metin = AB
%c	Karakterler için kullanılır	printf "Karakter = %c" 'A'	Karakter = A
%b	Mantıksal değerler için kullanılır	printf "Mantıksal doğru = %b" true	Mantıksal doğru = true
%O	.NET'de herhangi bir nesenin ToString() fonksiyonunun ürettiği değer basmak için kullanılır	printf "Tarih ve Saat = %O" 'A'	Tarih ve Saat = 9/24/2017 4:48:00 PM
%A	Herhangi bir değeri basmak için kullanılır	printf "Tarih ve Saat = %A" 'A'	Tarih ve Saat = 9/24/2017 4:49:09 PM

Bu bölümde 01_01_00_a.png nolu referans resim kullanılacak.

```
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 basamaklı 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 yerine bir tane değer verilmiş
printf "Sayı değerileri %d ve %d" 1 2 // Doğru
```

Bilgi Kutusu (BİLGİ): Terminale metin yazdırmak için .NET standard 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.

1.2 Kısa F# Tarihçesi

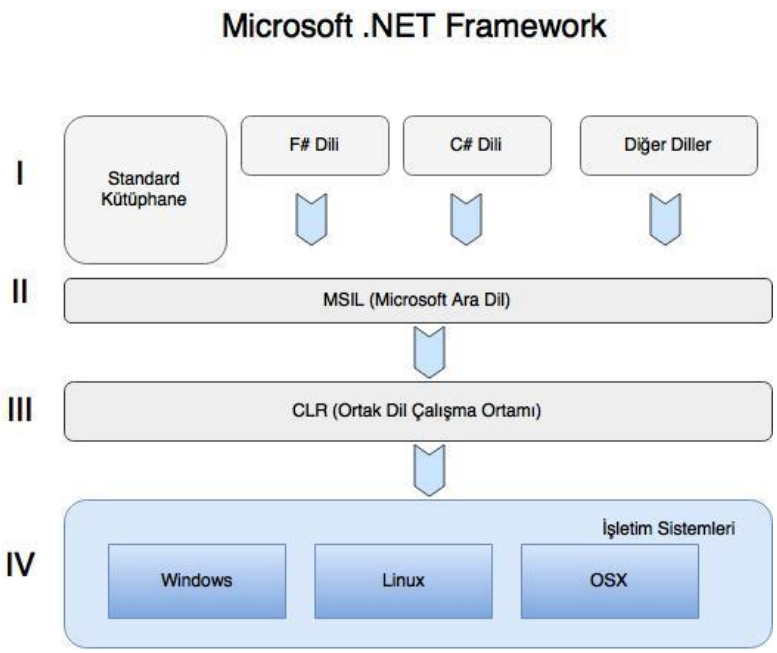
F#, Türkçe **efşarp** olarak telafuz edilen, yabancı kaynaklarda **FSharp** olarak da rastlayabileceğiniz, yordamsal (imperative) ve bildirimsel (declarative) programlama yaklaşımlarının her ikisini de destekleyen çok yönlü (multi paradigm) ve fonksyionel bir programlama dilidir.

Bilgi Kutusu (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, standard 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.



Bu bölümde 01_01_00.jpg nolu referans resim kullanılacak.

F#'ı geliştiren ekibin kurucusu olan Don Syme F#'ın nasıl ortaya çıktığını şöyle anlatıyor.

> .NET platformunun vizyonunda başlangıçtan itibaren birden fazla programlama dilinin desteklenmesi önemli bir hedef olarak yer alıyordu. 1998 yılında, programlama dilleri ile ilgili araştırma grubumdan 10 kişi ile birlikte Microsoft'a dahil olduktan sonra, Project 7 kod adlı projeyi başlatan James Plamondon bizimle irtibata geçti. Project 7, yedi adet akademik ve yedi adet de genel amaçlı programlama dilinin .NET üzerinde deneysel olarak geliştirilmesini hedefleyen bir projeydi. Project 7 ile farklı programlama dillerini destekleyebilmek için .NET'in sağlaması gereken mekanizmalar ve esneklikler araştırılacak ve sonuçlar .NET'e yansıtılacaktı.
>
> .NET'in Generic'leri üzerinde çalışırken elde ettiğim tecrübeyle ML benzeri fonksiyonel bir programlama dilinin .NET'i destekleyip desteklemeyeceğini araştırmak için ".NET için Haskell" üzerinde çalışmaya başladım. Bu çalışmada önemli gelişmeler sağlamamıza rağmen Haskell ile .NET arasındaki ciddi yaklaşım farklılıkları nedeni ile çalışmayı durdurduk.

Haskell deneyinin başarısız olması sonrasında Don Syme ve ekibi 2000'li yılların en popüler ML varyantı olan OCaml dilini .NET'e taşıma çabalarına yoğunlaştılar. 2005 yılında çalışmalarını tamamlayan ekip OCaml temelli F#'ın ilk versiyonunu yayınladı. OCaml ve F#'ın birbirlerine ne kadar benzediğin görmek için aşağıdaki faktöriyel hesaplama kodunu sırasıyla F# ve OCaml yorumlayıcılarında çalıştırabilirsiniz.

```
(* 01_1_01.fsx *)

let rec fact x = if x <= 1 then 1 else x * fact (x - 1)
fact 5
```

Bilgi Kutusu (BİLGİ): F# kodunu .NET Fiddle ile OCaml kodunu ise OCaml Pro adreslerinden online çalıştırabilirsiniz. OCaml kodunu denerken her bir satırın sonuna **;;** eklemeyi unutmayın!

F#, 2017 yılı itibariyle 4.1 versiyonuna ulaşımış, arkasında Microsoft gibi dev bir şirketin bulunduğu açık kaynak kodlu fonksiyonel bir programlama dili olarak varlığını sürdürmektedir. .NET'in çalıştığı platformların çeşitliliği arttıkça F#'ın ulaştığı kitleler ve farklı alanlardaki popülerliği de artmaktadır.

F# versiyon tarihçesini ve diğer ayrıntıları aşağıdaki çizelgeden inceleyebilirsiniz.

Versiyon	Tarih	Platform Desteği	.NET Versiyonu	Geliştirme Araçları
F# 1.x	2005 - Mayıs	Windows	.NET 1.0 - 3.5	Visual Studio 2015, Emacs
F# 2.0	2010 - Nisan	Windows, Linux, OSX	.NET 2.0 - 4.0, Mono	Visual Studio 2010, Emacs
F# 3.0	2012 - Ağustos	Windows, Linux, OSX, JavaScript,GPU	.NET 2.0 - 4.5, Mono	Visual Studio 2012, Emacs, WebSharper
F# 3.1	2013 - Ekim	Windows, Linux, OSX, JavaScript,GPU	.NET 2.0 - 4.5, Mono	Visual Studio 2013, Emacs, WebSharper, MonoDevelop, Xamarin Studio, CloudSharper
F# 4.0	2015 - Temmuz	Windows, Linux, OSX, JavaScript,GPU	.NET 2.0 - 4.5, Mono	Visual Studio 2013, Emacs, WebSharper, MonoDevelop, Xamarin Studio, CloudSharper
F# 4.1	2017 - Mart	Windows, Linux, OSX, JavaScript,GPU	.NET 3.5 - 4.6.2, .NET Core, Mono	Visual Studio 2017, Ionide, Visual Studio Code, Atom, Rider, Web Sharper, Visual Studio for Mac

Bu bölümde 01_01_01.png nolu referans resim kullanılacak.

Bilgi Kutusu (BİLGİ):

- F# ile ilgili daha ayrıntılı bilgilere fsharp.org üzerinden erişebilirsiniz.
- F#ın kodunu incelemek için ise F# GitHub deposuna başvurabilirsiniz.

1.3 Neden F#?

Farklı dillerde tecrübesi olan bir programcı olarak yeni bir programlama dili öğrenmeye başladığınızda bu dili zaten bildiğiniz diğer diller ile karşılaştıracaksınız. İlk defa bir programlama dili öğreniyorsanız yaptığınız tercihin size uygun olup olmadığına bir an önce karar vermek isteyeceksiniz.

Bu bölümde F# öğrenmeniz için sizi motive edeceğini umduğum bazı dil özelliklerini kod örnekleri ile ele alıyoruz. Göreceğiniz F# kodlarını bu aşamada tam olarak anlamayabilirsiniz. Bu nedenle, şimdilik kodları anlamaya değil kodlardaki zerafet ve şıklığa odaklanarak F#'ın size sağlayacağı katkıları ve karşılaşacağınız zorlukları analiz etmenizi öneriyorum.

Az Seremonili Söz Dizimi

F#, sade ve düşük seremonili bir söz dizimine (syntax) sahiptir. F#'da süslü parantezlere, noktalı virgüllere ve normal parantezlere nadiren ihtiyaç duyulur. F#'da kod alanları (global alan, modül, fonksiyon ve tip tanımı gibi girintiler (indentation) kullanılarak tanımlanır.

Aşağıdaki kod örneğinde // simgesi ile belirtilen yorum satırlarının hemen altındaki kod satırlarında bahsettiğimiz özellikleri görebilirsiniz.

```
(* 01_1_02.fsx *)

// Süslü parantez, parantez veya noktalı virgüle ihtiyacınız yok
// Kare fonksiyonu tanımı
let kare x = x * x

// Liste tanımlamak çok basit ve tek satır
// 1 ile 10 arasındaki sayıları barındıran liste
let sayılar = [1..10]

// Tek satırda listedeki sayıların karesini alıp yeni bir liste
üretebilirsiniz
let kareler = sayılar |> List.map kare

// Girintiler ile belirlenen kod blokları
let tekMiÇiftMi x = // Fonksiyon tanımı başlangıcı
    // Fonksiyonun kod alanı başlangıcı
    match x with
    | a when a <= 0 -> failwith "Değer sıfırdan büyük olmalı"
    | a when a % 2 = 0 -> true
    | _ -> false
    // Fonksiyonun kod alanı sonu

// Global alanda fonksiyon çağırısı
tekMiÇiftMi 12
```

Sade ve Şık Tip Tanımları

Kodun çözeceği problemin modellenmesi aşaması en önemli yazılım geliştirme aktivitelerinden birisidir. Modelleme aktivitesi UML dili ve araçları ile yapılabileceği gibi sadece kod yazılarak de yapılabilir. Kod yazarak yapılan modelleme bir taşla iki kuş vurmamızı sağlar. Şöyle ki;

- Hem problemi oluşturan parçaları probleme özel tipler ile tanımlarız
- Hem de değerli zamanımızı sadece model olarak kullanılabilecek bir çıktı için değil aynı zamanda çalıştırabilir kod üretmek için harcamış oluruz

Teoride tüm programlama dilleri modelleme için kullanılabilir. Ancak, kullanılan dilin sunduğı yapılar (tip, sınıf vs) yukarıda bahsettiğimiz avantajlar için katlanabileceğimizden daha karmaşık veya hantal olabilir. F#, bu karmaşa ve hantallıktan uzak bir dildir, çünkü sunduğu tip tanımlama yapıları sadelik ve şıklık konusunda oldukça iddialıdır.

Değer grupları (tuple), kayıt (record) ve ayrışık bileşimler (discriminated union) F#'daki temel tip tanımlama yapılarıdır.

```
(* 01_1_03.fsx *)

// Farklı tipte birden fazla değer barındırabilen basit tipler (tuple)
let çocuk = ("Arda","Özgür",10)
let ad,soyad,yaş = çocuk // değerleri çözümleme

// Daha yapısal tipler (record)
type Kişi = {Ad:string;Soyad:string}

// Yeni kişi kaydı oluşturma
let arda = {Ad="Arda";Soyad="Özgür"}
let kuzey = {Ad="Kuzey";Soyad="..."}

// Daha karmaşık tip tanımları (discriminated union)
type Kullanıcı =
    | Öğrenci of Kişi
    | Yönetici of Kullanıcı list

// Öğrenci ve yönetici oluşturma
let öğrenci1 = Öğrenci arda
let öğrenci2 = Öğrenci kuzey
let yönetici = [öğrenci1;öğrenci2]
```

Güçlü Tip Sistemi

Programlama dilleri sınıflandırmasında dinamik tipli diller ve statik tipli diller şeklinde genel bir ayrım yapılır. Statik tipli dillerde değişkenler, metod parametreleri ve metodun dönüş değeri için tip tanımı yapılması zorunludur. Tip uyumu derleme anında sıkı bir şekilde kontrol edilir. Dinamik tipli dillerde ise herhangi bir tip tanımı yapılmasına gerek kalmadan değişkenler, parametreler ve metodlar tanımlanabilir. Tip kontrolü derleme anında değil çalışma anında yapılır.

Her iki yaklaşımın da çok seveni olduğu gibi bir o kadar da nefret edeni vardır. Bu iki yaklaşımın sağladığı avantajları ve dezavantajlar kitabımızın kapsamı dışında olduğu için ayrıntılara yer vermiyoruz. Ancak, benim size tavsiyem bu konudaki fanatik tarışmalardan uzak durarak elinizdeki problemi en iyi şekilde çözeceğini düşündüğünüz yaklaşıma göre tasarlanmış dilleri kullanmanızdır.

F# derleyici seviyesinde statik tipli diller gibi davranırken kod yazımı sırasında dinamik tipli diller gibi davranır. Bunun iki anlamı vardır;

1. Kod yazarken değer ifadeleri ve fonksiyon tanımlarında parametre tiplerinin verilmesi çoğunlukla zorunlu değildir (dinamik dillerdeki gibi).
2. Derleyici biraz akıllı davranarak derleme sırasında tip uyumluluğunu kontrol edip hataları yakalar.

F#'ın bu yaklaşımının arkasındaki mekanizmaya **tip çıkarsama** (type inference) denir. Tip çıkarsama sayesinde tip bildirimlerine ihtiyaç duymadan daha kısa ve okunaklı kod yazabiliriz. Buna ilave olarak yazılan kodun tip uyumluluğu anlamında güvenli olması derleyici tarafından garantilenir.

```
(* 01_1_04.fsx *)

let tamSayı = 1 // int
let metin = "Neden F#" // string
let pi = 3.14 // float
let evetHayır = true // bool

// Kare alma fonksiyonu. Girdi parametresi ve çıktının int olduğu
çıkarsanır
let kare x = x * x
let sonuç1 = kare 12
//let sonuç2 = kare 3.14 // Hata girdi parametresi int değil

// Ondalık basamaklı sayılar için kare fonksiyonu. Girdi parametresi ve
çıktı olarak float olacağını belirttik
let kare2 (x:float) : float = x * x
let sonuç3 = kare2 3.14
//let sonuç4 = kare2 3 // Hata girdi parametresi float değil
```

```
// Kişi ve Çalışan tipinde kayıt tanımları
type Çalışan = {Ad:string;Soyad:string}
type Kişi = {Ad:string;Soyad:string}


// çocuk ve baba değer ifadelerinin tipini belirtmedik buna rağmen tipinin
Kişi olduğu çıkarsanır
let çocuk = {Ad="Arda";Soyad="Özgür"}
let baba = {Ad="Ali";Soyad="Özgür"}

// anne değer ifadesinin Çalışan tipinden olduğunu biz ifade ettik
let anne = {Çalışan.Ad="Seniha";Soyad="Özgür"}
```

Tip çıkarsama mekanizması her zaman tutarlı sonuç üretir.Ancak, tip çıkarsama bazen ne ifade etmek istediğimizi net olarak belirtmediğimiz için varsayımlar yapar. Bu varsayımların sonucunda ise bizi memnun etmeyen hatalı tipler çıkarsanır. Yukarıdaki örnekte yer alan

let seniha = {Çalışan.Ad="Seniha",Soyad="Özgür"} ifadesini

let seniha = {Ad="Seniha",Soyad="Özgür"}

şeklinde yazarsak **Kişi** tipi **Çalışan** tipi tanımından sonra yapıldığı için **anne** değer ifadesinin tipinin **Kişi** olduğu çıkarsanır. Bu durum **anne** değeri tanımında alanlardan ilkinin önüne alanın hangi tipe ait olduğunu **Çalışan.Ad="Seniha"** şeklinde belirterek engellenebilir. Bu yöntem ile F# derleyicisine bir ipucu verilerek tip çıkarsama sırasında varsayım yapması önlenir.

Sade ve Yetenekli Veri Yapıları

Çok genel bir tanıma göre yazılım programları akış kontrolü, veri alma/verme ve işleme kabiliyeti olan akıllı görünümlü otomasyon sistemleridir. Bu basit tanıma istinaden yazılan kodun önemli bir miktarının fonksiyonlar arasında, tipler arasında, modüller arasında veya diğer yazılımlar ile veri alış verişini sağlayan ifadelerden oluştuğu söylenebilir.

F#, bu basit tanımda yer verilen işlemler için hem dil seviyesinde hem de standard kütüphanesinde performanslı çalışan ve kullanımı kolay yapılar sunar. Aşağıdaki örnekte F#'ın temel veri yapılarından olan liste, dizi ve sekans (silsile) tiplerinin kullanımını inceleyebilirsiniz.

```
(* 01_1_05.fsx *)
open System

// 1 ile 5 arasındaki sayıları barındıran liste
let list1 = [1;2;3;4;5]

// 6 ile 10 arasındaki sayıları barındıran liste
let liste2 = [6..10]

// 12 ile 20 arasındaki çift sayıları barındıran liste
let liste3 = [12..2..20]
```

```
// 1 ile 5 arasındaki sayıları barındıran dizi
let dizi1 = [|1;2;3;4;5|]

// 6 ile 10 arasındaki sayıları barındıran dizi
let dizi2 = [|6..10|]

// 12 ile 20 arasındaki çift sayıları barındıran dizi
let dizi3 = [|12..2..20|]

// 1 ile int tipinin en büyük değeri arasındaki sayıları barındıran
sekans/silsile
let sayılar4 = seq{1..System.Int32.MaxValue}
```

Bilgi Kutusu (NOT): seq (sekans veya silsile) fiziksel bellek kapasitesini izin verdiği ölçüde sınırsız sayıda elemanı barındırabilen standard bir tiptir. **seq** yüksek boyutlu veri işlemlerinde kullanılması önerilen en yüksek performanslı tiplerden birisidir.

Sade ve şık veri yapılarına ilave olarak F# standard kütüphanesindeki **List**, **Seq** ve **Array** modülleri onlarca fonksiyonu hazır olarak kullanımınıza sunar.

Aşağıdaki örnekte **List** modülü içinde yer alan bazı fonksiyonlarını kullanımı gösterilmektedir.

```
(* 01_1_06.fsx *)

// 1 ile 100 arasındaki değerleri barındıran liste
let liste = [1..100]

// List.map
// Listedeki değerlerin ondalık değerlere çevirip ve yeni bir liste
oluştur
let ondalıkSayıListesi = liste |> List.map (fun x -> float(x))

// List.average
// Listedeki değerlerin ortalaması
let ortalama = ondalıkSayıListesi |> List.average

// List.choose
// Listedeki 50'den büyük değerler seçilir
let büyükSayılar = liste |> List.choose (fun x -> if x > 50 then Some x
else None)

// List.chunkBySize
// Listeyi üçlü gruplar halinde sayıları barındıran listeye çevir
let üçlüGruplarListesi = liste |> List.chunkBySize 3

// List.filter
// Listedeki 50'den küçük sayıları filtrele ve yeni bir liste oluştur
let küçükSayılar = liste |> List.filter (fun x -> x <=50)
```

```
// @ iki listeyi ekleme operatörü
// :: listenin başına eleman ekleme operatörü
// 200 ile 300 arasındaki sayıları barındıran liste
let liste2 = [200..300]

// liste ve liste2'yi birleştir ve yeni bir liste oluştur
let liste3 = liste @ liste2


// liste3'ün başına 0 değerini ekle
let liste4 = 0 :: liste3

// liste4'ün sonuna 301 ekle
let liste5 = liste4 @ [301]

// List.iter ve List.iteri
// liste5'in elemanları üzerinde tek tek ilerle ve her bir elemanı
kullanarak değerini ekrana bas
liste5 |> List.iter (fun x -> printfn "Değer = %d" x)

// liste5'in elemanları üzerinde tek tek ilerle ve her bir eleman ve
elemanın indeksini kullanarak pozisyonunu ve değerini ekrana bas
liste5 |> List.iteri (fun i x -> printfn "Değer %d = %d" i x)
```

Bilgi Kutusu (BİLGİ): |> operatörü **pipe forward (ileri aktarım)** olarak adlandırılan ve **let (|>) x f = f x** şeklinde tanımlanan özel bir ikili (binary) operatördür. Bu tanımdaki (|>) ifadesi fonksiyonun adı, x normal bir değer ve f de bir fonksiyondur. Normalde **f x** şeklindeki yapılması gereken fonksiyon çağırıları bu operatör kullanılarak **x |> f** şeklinde de yapılabilir.

Eş zamanlı ve paralel işlemler

Sahip olunan kaynakları en verimli şekilde kullanıp makul sürede sonuç üretebilmek bulut ekonomisinin en önemli gelişim alanlarından birini oluşturmaktadır. Programlama dillerinin sunduğu eş zamanlı ve paralel işlem yapıları birim zamanda işlenen veriyi ve kurulan bağlantı miktarını arttırdığı için önemli ilerlemelerin ve keşiflerin kapılarını açar. Örneğin, sosyal platformlar ve IoT (nesnelerin interneti) tarafından üretilen büyük verinin düşük birim maliyetler ile işlenebilmesi yapay öğrenme ve davranış analitiği alanında geniş bir yelpazede farklı uygulamaları mümkün kılmaktadır.

F#, eş zamanlı (asenkron) ve paralel işlemler için kullanımı basit dil yapıları, standard kütüphane fonksiyonları ve mesaj tabanlı işlem yapabilmek için çeşitli mekanizmalar sunar.

```
(* 01_1_07.1.fsx *)
(*
    async ifade ile değerleri eş zamanlı olarak terminale yazdırma
*)
open System
open System.Net
open Microsoft.FSharp.Control.CommonExtensions

// Değeri ekrana yazdıran asenkron ifade
let ekranaYazdır değer =
    async {
        printfn "Değer %d" değer
        }

// Yazdırılacak değerler
let sites = [0..10]

sites
|> List.map ekranaYazdır   // Eş zamanlı görevleri oluştur
|> Async.Parallel          // Görevleri paralel hale getir
|> Async.RunSynchronously  // Görevleri
```

F#'da herhangi bir işlemi asenkron hale getirmek için **async** ifadeler kullanılır. Örneğimizdeki **ekranaYazdır** ifadesi arka planda eş zamanlı çalıştırılabilecek bir ifadedir.

```
(* 01_1_08.fsx *)
(*
    Fibonacci sayılarının paralel olarak hesaplanması
*)

// Fibonacci sayısını hesaplayan fonksiyon
let rec fib n =
    match n with
    | n when n=0 -> 0
    | n when n=1 -> 1
    | n -> fib(n - 1) + fib(n - 2)

// Paralel çalışacak görevleri oluştur
let işlemler = Async.Parallel [ for i in 0..10 -> async {  return fib i }
]

işlemler
|> Async.RunSynchronously // Görevleri çalıştır
|> Array.iteri ( fun i x -> printfn "fib(%d) = %d" i x) // Sonuçları
ekrana yazdır
```

F# standard kütüphanesindeki **Async** modülünde yer alan **Parallel** ve **RunSynchronously** gibi fonksiyonlar kullanarak arka planda eş zamanlı çalışabilen görevler oluşturulup çalıştırılabilir.

Bilgi Kutusu (BİLGİ): Async.RunSynchronously fonksiyonun adında bakıldığında görevleri senkron yani ardı ardına çalıştıracakmış gibi bir izlenim oluşabilir. Ancak bu fonksiyon arka planda çalışacak tüm görevleri eş zamanlı olarak başlatıp hepsi tamamlanana kadar çağırının yapıldığı thread bekletmek için kullanılır. Program akışı ancak ve ancak tüm arka plan görevleri tamamlandığına bir sonraki satırdan devam eder. Eğer arka plan görevlerinin tamamlanması beklenmeden akışın devam etmesi istenirse **Async.Start** vey **Async.StartImmediate** başlatma fonksiyonları kullanılmalıdır.

Bu iki yapıya ilave olarak F# standard kütüphanesi içindeki **MailboxProcessor** tipi ile mesaj tabanlı ve asenkron kuyruk (queue) kullanımını gerektiren işlevler de geliştirilebilir.

```
(* 01_1_09.1.fsx *)
// MailboxProcessor
// Ajanı oluştur
let ajan = MailboxProcessor.Start(fun kuyruk ->
    let rec mesajDöngüsü() =
        async{
            let! msg = kuyruk.Receive()
            printfn "Gelen Mesaj: %s" msg
            do! mesajDöngüsü()
            }
    mesajDöngüsü()
)

// Ajana mesaj gönder
ajan.Post "F# ile Fonksiyonel Programlama"
```

Bilgi Kutusu (BİLGİ): let! ifadesi asenkron fonksiyon çağırılarının sonuçlarını değer ifadelerine bağlamak için kullanılır. **let!** kullanıldığında asenkron işlem tamamlanana kadar kod akışı bir sonraki satırdan devam etmez. Asenkron fonksiyon çağırılarında sadece **let** kullanıldığında ise fonksiyon çalıştırılmaz bunun yerine asenkron çalışabilecek bir görev (Async<'T> tipinden) oluşturulur. Bu asenkron görev daha sonra **Async.RunSynchronously** veya **Async.StartImmediate** ile çalıştırılmalıdır.

Fonksiyonel Olmayan Yöntem Desteği

F# temelde fonksiyonel bir dildir. Ancak, .NET'i destekleyen diğer diller ile uyumluluğun sağlanması için fonksiyonel yaklaşıma ters düşen prosedürel ve nesne yönelimli yaklaşımlar da F# tarafından desteklenir.

```
(* 01_1_10.fsx *)
open System

// Değişken
let mutable sayı = 42
sayı <- 43

let dizi = [|1..100|]
// for ve  if/else yapıları
for i in dizi do
    if i % 2 = 0 then
        printfn "Çift Sayı = %d" i
    else
        printfn "Tek Sayı = %d" i
```

```
// yan etkili fonksiyon
// printfn'in yan etkisi terminale metin yazdırmasıdır
printfn "Sayının değeri = %d" sayı

// System.Int32 .NET'in sağladığı tam sayı tipidir
// Aşağıdaki ifade ile System.Int32 tipi için
// ÇiftMi isimli ek metod tanımı
type System.Int32 with
    member this.ÇiftMi() = this % 2 = 0

// System.Int32 tipinden iki sayı
let çiftSayı:System.Int32 = 12
let tekSayı:System.Int32 = 11

// sayıların çift olup olmadığını kontrolü
çiftSayı.ÇiftMi()
tekSayı.ÇiftMi()

// Nesne tabanlı programlama dillerindeki gibi sınıf tanımları
[<AbstractClass>]
type Şekil =
    abstract member Renk : string
    abstract AlanHesapla : unit -> float
```

Bu çoklu yaklaşım (multi-paradigm) sayesinde fonksiyonel olmayan diller (örneğin C# veya Java) ile tecrübesi olan yazılım geliştiriciler tarzlarını çok fazla değiştirmeden olabildiğince hızlı bir şekilde F# kullanmaya başlayabilirler. Ancak prosedürel yaklaşım F#'ın sağladığı avantajları ortadan kaldırır. Uzun vadede F#'ın sağladığı fonksiyonel yapı ve yöntemlere adapte olmanızı tavsiye ediyorum.

Geniş Uygulama Yelpazesi

F# uzun bir geçmişe sahip fonksiyonel bir dilidir. Başarı hikayelerine bakıldığında enerji, sağlık, finans, sigortacılık, DNA araştırmaları, akademik araştırmalar, genel amaçlı web ve mobil uygulamalar, orta katman uygulamaları, veri analizi ve görselleştirme, kara para aklama tespit uygulamaları, analitik uygulamalar gibi bir çok sektörde kullanıldığını görebilirsiniz.

Şimdi sıra sizde! Siz de F#'ı öğrenerek kendi sektörünüze özgü programlar geliştirip başarı hikayeleri sayfasında yer alabilirsiniz.

Aktif Geliştirici Topluluğu

F#, Microsoft tarafından geliştirilen bir dil olmasına rağmen açık kaynak olarak yayınlanmaktadır. Microsoft dilin geliştirilmesine sadece tam zamanlı iş gücü katkısı yapar, bunun dışında dilin tasarımı ve yol haritası ile ilgili kararlar F# geliştiricileri ve kullanıcılarının oluşturduğu topluluk tarafından demokratik bir şekilde alınır ve uygulanır. Microsoft çalışanı olan bir F# geliştiricisi ile bağımsız bir F# geliştiricisinin dile katkı yapma fırsatları eşittir.

Siz de F#'ı GitHub deposunandan klonlayıp kod ve dokümantasyon katkısı yapabilir veya yeni özellik taleplerinizi F# topluluğunun tartışmasına ve değerlendirmesine sunabilirsiniz.

Hazır Paketler

F# bir .NET dili olduğu için .NET için geliştirilmiş tüm paket kütüphanelerini Microsoft'un resmi paket yayınlama platformu olan **NuGet** üzerinden indirerek kendi programlarınızda kullanabilirsiniz.

Bilgi Kutusu (İPUCU): NuGet uygulamasına alternatif olarak açık kaynak kodlu **Paket** ile de projelerinizde kullanmak için paket kütüphanelerini indirip yönetebilirsiniz.

1.4 Fonksiyonlara Matematiksel Bakış

Fonksiyonel programlamanın temeli matematiksel fonksiyonlar ve fonksiyonların bazı özellikleri üzerine inşa edilmiştir. Matematiksel açıdan en genel **fonksiyon** tanımı aşağıdaki gibi yapılır.

X ve Y iki küme, f⊂X×Y bir bağıntı olsun. Aşağıdaki koşullar sağlanırsa f bağıntısına bir fonksiyon denir: 1. ∀x∈X, ∃y∈Y:(x,y)∈f, 2. (x, y),(x,y′)∈f⇒y=y′

Burada X'e tanım kümesi, Y'ye ise değer kümesi denir. Tanımından da anlaşılacağı gibi fonksiyon, tanım kümesindeki her elemanı, değer kümesindeki tek bir elemanla eşleştiren bir bağıntıdır. Bu yüzden fonksiyonlarda xfy veya (x,y)∈f gösterimi yerine y=f(x) gösterimi kullanılır. Bir fonksiyona bazen dönüşüm de denir. Eğer f, X'den Y'ye bir fonksiyon ise bu durum f:X→Y ile ya da X→fY ile gösterilir.

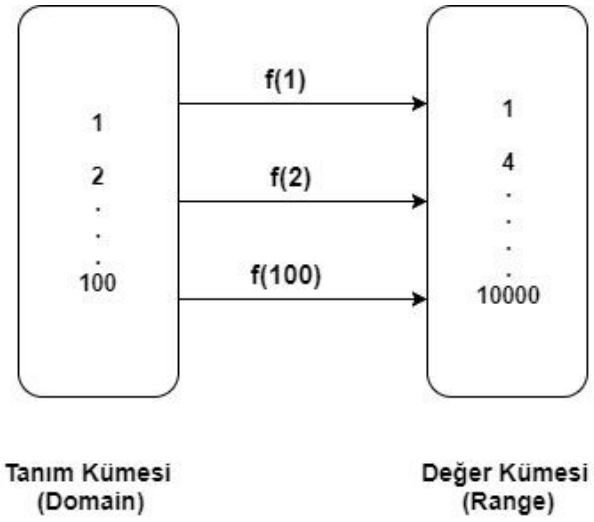
Yukarıdaki tanımda belirtilen 1. koşuldaki **∀x∈X** ifadesini "X kümesinin elemanı olan tüm x değerleri", **∃y∈Y** ifadesini ise "Y kümesinin elemanı olan bir y değeri" şeklinde okuyabilirsiniz. **∀** ve **∃** sembolleri matematikte nicelik/miktar belirten sembollerdir, **∀** sembolü **tüm** ve **∃** sembolü de **bir** anlamında miktar belirtir.

Bu tanımda yer alan diğer iki sembolden **∈** sembolü bir değerin bir kümenin elemanı olduğunu ifade eder, **⊂** sembolü ise **alt küme** anlamına gelir. Tanımımıza göre (x,y) değer çiftinin f fonksiyonunun üreteceği sonuç kümesinin bir alt kümesi olduğunu söyleyebiliriz.

Tanımın ikinci koşulu olan " (x,y),(x,y′)∈f⇒y=y′ " ifadesini ise şöyle yorumlarız; f fonksiyonu, X değer kümesinin bir x elemanını Y kümesinin y ve y′ şeklinde iki elemanı ile eşleştiriyorsa y ve y′ değerleri birbirine eşittir. Başka bir deyişle, f fonksiyonu X tanım kümesinin elemanı olan bir x değerini her zaman Y değer kümesinin bir elemanı olan aynı y değeri ile eşleştirir.

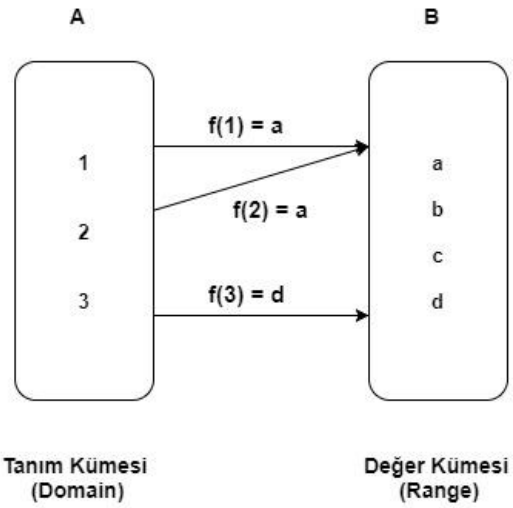
Şimdi gelin bu fonksiyon tanımını görselleştirerek basit bir örnek ile somutlaştıralım.

f(x) = x * x şeklinde bir fonksiyon tanımı olsun. Bu fonksiyon girdi olarak verilen x değerinin karesini hesaplar. Daha matematiksel bir şekilde ifade edecek olursak; bu fonksiyon doğal sayılar kümesinin elemanı olan tüm **x** değerlerini yine doğal sayılar kümesinin elemanı olan bir **x*x** değeri ile eşleştirmektedir.



Bu bölümde 01_01_01.jpg nolu referans resim kullanılacak.

Yukarıdaki şekilde yer alan **tanım kümesi** ve **değer kümesi** kavramları önemlidir, zira fonksiyonları tanım kümesindeki elemanları değer kümesindeki elemanlar ile eşleştiren dönüşümler şeklinde de ifade ederiz.



Bu bölümde 01_01_02_drawio.jpg nolu referans resim kullanılacak.

Yukarıdaki örnekte * Tanım Kümesi A : A{1,2,3} * Değer Kümesi B : B{a,b,c,d} * Görüntü Kümesi : f(A) = {a,d}

f fonksiyonunu da f(A) = {(1,a),(2,a),(3,d)} şeklindeki eşlemelerin kümesi olarak tanımlarız.

1.5 Fonksiyonların İlginç Özellikleri

Matematiksel fonksiyonların fonksiyonel programlama dillerinin yapısını yakından etkileyen belirleyici iki önemli özelliğinden bahsedebiliriz, bunlar

1. Fonksiyonlar tanım kümesindeki bir elemanı her zaman değer kümesindeki aynı eleman ile eşleştirir
2. Fonksiyonların yan etkileri yoktur

f(x) = x * x şeklindeki fonksiyon tanımını örnek olarak ele alırsak, bu fonksiyonun tanım kümesindeki 2 değerini değer kümesindeki 4 değeri ile, 3 değerini de 9 değeri ile eşlediğini söyleriz. Bu fonksiyonun tanım kümesindeki 2'yi değer kümesinde 4'ten farklı bir değer ile eşlemesi mümkün değildir. Fonksiyonlar **aynı girdi** için hep **aynı çıktıyı** üretir.

f(x) = x * x fonksiyonu matematiksel tanımına uygun olarak F# ile aşağıdaki gibi ifade edilebilir.

```
(* 01_2_01.fsx *)
let f (x) =
    match x with
    | 1 -> 1
    | 2 -> 4
    | 3 -> 9
    | _ -> -1
```

Dikkat ederseniz fonksiyonları bu noktaya kadar hep *eşleme yapan birer dönüşüm* olarak tanımlamaya özen gösterdik. Eğer fonksiyonel olmayan programlama dilleri ile tecrübeniz varsa fonksiyonların veya metodların hesaplama yapmak için kullanıldığını düşünüyor olabilirsiniz. Ancak yükarıdaki **f(x) = x * x** örneğinde de görebileceğiniz gibi fonksiyonlar aslında herhangi bir hesaplama yapmazlar, fonksiyonlar basitçe iki kümenin elemanlarını birbirleri ile eşlerler. Fonksiyonları, programcı bakış açısıyla, herhangi bir hesaplama yapmayan basit birer switch/case (C,C++, Java, C#, JavaScript gibi dillerin hepsinde olan koşullu dallanma yapısı) kod bloğu olarak düşünebilirsiniz.

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 paralele çağırıları mümkün kılar.
- Bir fonksiyonu çıktısına ihtiyaç duyduğunuz anda yani sonrada değerleyebilirsiniz (lazy evaluation). Fonksiyonel olmayan programlama dillerinde program akışı bir fonksiyona geldiği anda o fonksiyon hemen çalıştırılır. Fonksiyonun dönüş değeri de bir bellek konumunda saklanabilir. Fonksiyonel programlama dillerinde ise program akışı bir fonksiyona geldiğinde eğer dönüş değerine hemen ihtiyacınız yoksa fonksiyon değerlemesini geciktirebilirsiniz. Buna **sonradan değerleme** (lazy evaluation) denir. Fonksiyonların birinci özelliği sonradan değerlemeyi mümkün kılar çünkü bir fonksiyonu ne zaman değerlerseniz değerleyin aynı girdi için her zaman aynı çıktı üretilecektir.
- Fonksiyonlarınıb dönüş değerlerini daha sonra kullanılmak üzere bellemesini sağlayabilirsiniz. Fonksiyonel programlama dillerinde bu özelliğe **belleme**(memoization) denir. Belleme davranışı doğrudan fonksiyon tanımında ifade edilebilir. Fonksiyon eğer daha önce bellediği bir eşleştirme işlemini yapmak üzere çağırılırsa bu işlemi gerçekten yapmadan bellenen sonucu döndürür. Belleme de fonksiyonların birinci özelliği sayesinde mümkündür.
- Birden fazla fonksiyon istenen sırada değerlenebilir(evaluate). Fonksiyonlar değerlendiğinde girdi parametresi değeri değişmediği için (girdi değeri bozulmadığı için) döndüş değeri de değişmez. Bu davranış fonksiyonların ikinci özelliğinin (yan etkisinin olmaması) bir sonucudur.

Değerleme Sırası Önemli Mi Değil Mi?

Fonksiyonların ikinci özelliğine istinaden fonksiyonları istediğimiz sırada değerleyebileceğimizi ve sonucun değişmeyeceğini söylemiştik. Ancak matematiksel olarak **f(g(x)) = g(f(x))** önermesi her zaman doğru değildir. Bu önerme sadece bazı özel **f** ve **g** fonksiyonları için doğru olabilir (örneğin birim fonksiyon). Bu özel fonksiyonlar dışındaki fonkisyonlar için **f(g(x)) ≠ g(f(x))** önermesi geçerlidir.

Fonksiyonların çalıştırma sırasını önemli olduğuğunu aşağıdaki örnek programımızda da hızlıca görebiliriz. Sıralama değiştirildiğinde işlem sonucu da değişir.

```
(* 01_2_04.fsx *)
let f(x)  = x + 1 // bir arttırma fonksiyonu tanımı
let g(x) = x * x // kare alma fonksiyonu tanımı

printfn "Sonuç f(g(1)) = %d" (f(g(1))) // Sonuç f(g(1) = 2
printfn "Sonuç g(f(1)) = %d" (g(f(1))) // Sonuç g(f(1) = 4
```

Fonksiyonel programlama terminolojisinde değerleme (evaluate) ve çalıştırma (execute) aynı anlama gelmez. Değerleme sırası derleyici seviyesinde geçerli bir kavramdır ve yazdığınız kodun çalıştırılma sırası ile doğrudan bir ilişkisi yoktur. Bu nedenle matematiksel ve programatik olarak yukarıdaki örnekteki **f(g(x))** ve **g(f(x))** çağırıları eş çağırılar değildirler. Bu nedenle fonksiyonel programlamada değerleme sırası önemli olmamakla birlikte çalıştırma sırası diğer tüm programlama yaklaşımlarında olduğu gibi çok önemlidir.

f(g(1)) ifadesi için derleyici iki farklı değerleme stratejisi kullanabilir. İlk değerleme **Normal Sıralı Değerleme** (Normal Order Evaluation) aşağıdaki adımlardan oluşur

```
// Normal Değerleme

f(g(1))
= g(1)+1  // f(x) = x + 1 olduğu için f(x) g(1) + 1 olarak değerlendi
= (1*1)+1 // g(1) -> 1*1 olarak değerlendi
= 1 + 1   // g(1) = 1 olduğu için ifade 1 + 1 olarak değerlendi
= 2
```

İkinci değerleme yaklaşımı **Uygun Sıralı Değerleme** (Applicative Order Evaluation) ise şöyle olur

```
f(g(1))
= f(1*1) // önce g(1) ifadesi değerlendi -> 1*1
= f(1)   // sonuç f(1)
= 1+1    // sonra da f(1) ifadesi değerlendi -> 1 + 1
= 2      // sonuç
```

Hangi değerleme yaklaşımı uygulanırsa uygulansın **f(g(1))** ifadesinin sonucu değişmez ve 2'ye eşittir.

Bilgi Kutusu (BİLGİ):

> **Normal Sıralı Değerleme** yapılırken bir fonksiyonun en soldaki ifadesi öncelikli olarak değerlenir. f(g(1)) ifadesinde en solda f fonksiyonu var ve f(x) = x + 1 olduğu için f(g(1)) ifadesi açılarak g(1) + 1 şeklinde yazılır. Programlama terminolojisinde buna **isimle çağırma** (call by name) da denir
>
> **Uygun Sıralı Değerleme** yapılırken en içteki fonksiyonun ifadesi öncelikli olarak değerlenir. f(g(1)) ifadesinde en içteki fonksiyon g fonksiyonu olduğu için g(1) ifadesi değerlendi ve f(g(1)) ifadesi f(1 * 1) olarak yazıldı. Programlama terminolojisinde buna **değerle çağırma** (call by value) da denir

Kullandığınız fonksiyonel programlama dilinin derleyicisi her zaman yukarıdaki değerleme yöntemlerinden sadece birini kullanabilir. Yazdığınız ifadelere veya derleyicinin çalıştırıldığı donanımın yeteneklerine göre iki değerleme yöntemini duruma göre değişimli olarak da kullanabilir.

Fonksiyonların İlginç Olmayan Özellikleri

Fonksiyonların ilginç özelliklerinin yanısıra pek de ilginç olmayan iki özelliğinden daha bahsedebiliriz.

1. Fonksiyonların girdi prametre değerleri ve döndürdükleri değerler değiştirilemez. Buna **değerin değişmezliği (immutability)** denir.
2. Fonksiyonların tek bir girdi parametresi ve tek bir dönüş değeri vardır

Bu iki özellik ilk başta çok önemli değilmiş hatta biraz da kısıtlayıcıymış gibi görünebilir. Ancak bu özellikler fonksiyonel programlama dillerinin tasarımını doğrudan etkiler. Örneğin F# derleyicisi yazdığınız tüm fonksiyonları tek bir giriş parametresi alan ve tek bir çıktı üreten birer fonksiyon olarak değerler. Benzer şekilde F#'da **let** ile tanımlanan değer ifadelerinin değerlerinin tanımlandığı andan sonra değiştirilmesine izin verilmez.

Bilgi Kutusu (BİLGİ) : F#'da **değişken (variable)** terimi yerine **değer ifadesi (value expression)** terimi kullanılır. Örneğin aşağıdaki a,b ve pi değer ifadeleri değişken değildir çünkü değerlerini bir defa tanımladıktan sonra farklı bir kod satırında değiştiremeyiz (*değişmezlik - immutability*)

```
(* 01_2_05.fsx *)
let a = 42
//a = 43 // Hata

let b = "F# ile Fonksiyonel Programlama"
//b = "F# ile fonksiyonel programlama" // Hata

let pi = 3.14
//pi = 3.0 // Hata
```

Ancak F#'da multi paradigm bir dil olması nedeni ile değeri değiştirilebilen (mutable) ifadeler tanımlamak da mümkündür. Bunun için **let mutable** bildirimi ve <- atama ifadesi kullanılır

```
(* 01_2_06.fsx *)
let mutable a = 42
printfn "a = %d" a

a <- 43 // Değer ifadesinin değerini değiştir
printfn "a = %d" a


let mutable b = "F# ile Fonksiyonel Programlama"
printfn "b = %s" b

b <- "F# ile fonksiyonel programlama" // Değer ifadesinin değerini
değiştir
printfn "b = %s" b
```

```
let mutable pi = 3.14
printfn "pi = %f" pi
pi <- 3.0 // Değer ifadesinin değerini değiştir
printfn "pi = %f" pi
```

Fonksiyonların Parametre Sayısı

Fonksiyonları matematiksel olarak tanımlarken tanım kümesindeki bir değeri değer kümesindeki bir değer ile eşleştiren dönüşümler olarak tanımlamıştık. Bu tanıma göre fonksiyonlar her zaman tanım kümesindeki tek bir değeri girdi olarak alıp değer kümesindeki tek bir değer ile eşleştirirler. Ancak matematikte birden fazla girdi alan ve girdiyi birden fazla değer ile eşleştiren çok boyutlu fonksiyonlar da tanımlanabilir.

```
// Tek girdi ve tek çıktı
f(x) = y

// Birden fazla girdi, birden fazla çıktı
f(u,v) = (u²-v, v²+u)

//Tek girdi birden fazla çıktı
f(t) = (cost(t), sin(t))
```

Matematiksel olarak çok parametreli ve çıktılı fonksiyonlar söz konusu olduğunda birden fazla boyutlu uzay ve bu uzaydaki noktaları ifade eden tanım ve değer kümeleri ile düşünmeye başlamalıyız. Ancak fonksiyonel programlama dillerinin modellenmesinde her zaman bir girdi ve bir çıktılı **tek boyutlu fonksiyonlar** kullanılır. Ancak, çok boyutlu uzaydaki noktalar iki değerli, üç değerli veya N değerli koordinatlar olarak gruplanıp bu gruplardan oluşan tanım ve değer kümeleri oluşturmak mümkündür. Bu şekilde tüm fonksiyonlar ilk tanımımıza uygun olarak tek girdi (koordinat) alan ve tek çıktı üreten dönüşümlere indirgenebilir.

F# derleyicisi fonksiyonları her zaman tek girdi ve tek çıktılı dönüşümler olarak işler. Bu durumda F#'da birden fazla girdiyi parametre olarak alan fonksiyon tanımlamak mümkün değil midir? Sorunun cevabı kısaca "Mümkündür". Bunun nasıl mümkün olduğunu ilerleyen bölümlerinde ayrıntıları ile bulabilirsiniz.

1.6 Fonksiyonel Programlama Nedir?

Fonksiyonel programlama, saf fonksiyonları (pure functions) ve değeri sonradan değiştirilemeyen ifadeleri (expressions) kullanarak yapılan kodlama faaliyetidir. Bazı kaynaklar fonksiyonel programlamayı fonksiyonların birinci sınıf vatandaş (first class citizen) olarak kabul edildiği kodlama faliyeti olarak da tanımlamaktadır. Fonksiyonel programlama bir araç veya dile bağlı değildir ve bir **yaklaşım** (paradigma) olarak değerlendirilir. Fonksiyonel olmayan programlama dilleri ile de (eğer dilin yapısı müsait ise) fonksiyonel programlama yaklaşımına ve ilkelerine uygun kod yazmak mümkün olabilir. Fonksiyonel programlama yöntemleri ile çoğu yazılım hatasının kaynağı olan paylaşılan program durumu (shared program state) ve yan etkilerden (side effect) arındırılmış kod yazmak mümkündür.

Fonksiyonel programlama yaklaşımına göre tasarlanmış programlama dilleri **bildirimsel (declarative)** diller sınıfında yer alır. Bildirimsel dilleri sınıfının karşısında da C, C++, Java, Pascal ve C# gibi **yordamsal (imperative)** diller yer alır.

Bilgi Kutusu (NOT): Programlama dilleri sınıflandırılırken bakış açısına bağlı olarak farklı yöntemler uygulamak ve farklı sınıflar tanımlamak mümkündür. Bildirimsel ve yordamsal dışında prosedürel diller, makina dilli, üst seviye diller, görsel diller, alana özgür diller gibi sınıflar da vardır.

Şimdi gelin basit bir F# kod parçası ile fonksiyonel kodun neye benzediğini deneyimleyelim.

```
(* 01_2_07.fsx *)

let liste = [1..10] // 1 ile 10 arasındaki sayıları barındıran liste
let kare x = x * x  // Bir sayının karesini alan fonksiyon tanımı

let sonuc = List.map kare liste // List modülü içindeki map fonksiyonu
printfn "Sonuç = %A" sonuc
// val sonuc : int list = [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
```

Yukarıdaki kod parçasında **list** isimli bir değer ifadesi ve **kare** isimli bir fonksiyon tanımı yapılıyor. **List.map kare liste** ifadesi ile de **List** modülü içindeki yüksek mertebeli **map** fonksiyon birinci parametresi **kare** fonksiyonu ikinci parametresi de **liste** olacak şekilde çalıştırılıyor.

Örneğimizdeki bazı kod satırlarının fonksiyonel programlama yöntemine uygunluğunu aşağıdaki gibi değerlendirebiliriz

- **kare** fonksiyonu saf bir fonksiyondur çünkü tanım kümesindeki her bir değer için her zaman aynı çıktıları üretir. İlave olarak fonksiyon içinde girdi veya çıktının değiri değiştirilmez
- **liste** ifadesinin değeri 1 ile 10 arasındaki sayılardır ve liste ifadesinin içeriği tanımlandığı andan sonra değiştirilemez
- **List.map** fonksiyonu yüksek mertebeli bir fonksiyondur çünkü **kare** fonksiyonunu parametre olarak kabul eder

Bilgi Kutusu (BİLGİ): Yüksek mertebeli fonksiyonlar başka bir fonksiyonu girdi parametresi olarak kabul eden fonksiyonlardır. Yukarıdaki örnekte kullanılan **List.map** fonksiyonu **kare** fonksiyonunu parametre olarak alabildiği için **yüksek mertebeli** (higher order) bir fonksiyondur.

Bildirimsel ve Yordamsal Programlama Yaklaşımları

F#, OCaml, Scala, Haskell gibi fonksiyonel programlama dilleri bildirimsel (declarative) diller olarak sınıflandırılır. C, C#, Java, Pascal ve Cobol gibi diller ise ana yaklaşımları nedeni ile yordamsal (imperative) diller şeklinde sınıflandırılır. Ancak, programlama dillerinin bu iki yaklaşıma göre hangi sınıfta yer aldığının belirlenmesi için çok net kriterler yoktur. Bazı diller (örneğin JavaScript, C# veya Java 8) destekledikleri programlama yapılarına göre her iki sınıfta da yer alabilmektedir. Tüm bu kriter belirsizliği ve karmaşasına reğmen bir programcı olarak bu iki sınıf arasındaki temel farkları bilmeniz hem F# öğrenirken hem de diğer diller ile çalışırken faydalı olacaktır.

Nasıl Yapılmalı? Yordamsal programlama dillerinde yazılan kod bir işlemin **nasıl** (how) yapılacağını tarif eder. Bu yüzden bu tür dillerin temel yapı taşları **tümcelerdir (sentence)**. Bu tümceler ile adım adım programın hangi işlemi **nasıl** yapması gerektiği tarif edilir ve bilgisayar bu adımları takip ederek programı çalıştırır. Bu sınıftaki dillere prosedürel diller de denir. Bu tür dillerde adım adım bir tarif söz konusu olduğu için genellikle akış kontrolü için **while** ve **for** gibi döngü yapıları, koşullu dallanma için **if/else** ve **switch** yapıları ve her bir adım sonrasında ulaşılan durumun takip edilmesi ve kayıt altına alınması için de **değişkenler** kullanılır.

Sonuç Ne Olacak? Bildirimsel programlama dillerinde ise yazılan kod bir işlemin nasıl yapılacağına değil işlem sonucunun **ne olacağına** (what) odaklanmıştır. Bu sınıftaki dillere fonksiyonel diller de denir. Bu tür dillerin temel yapı taşı **ifadelerdir** (expression). Bilgisayar, programlardaki bu ifadeleri çalıştırarak sonucun üretilmesini sağlar. Bildirimsel dillerde akış kontrolü için **öz yinelemeli fonksiyonlar** (recursive), koşullu dallanma için **yüksek mertebeli fonksiyonlar**(higher order functions) ve **match** benzeri yapılar kullanılır. Bildirimsel dillerde işlem sonucuna odaklanılır ve önceki adımlarda ulaşılan durumun takip edilmesi için değişkenlere ihtiyaç duyulmaz. Bu nedenle fonksiyonel dillerde doğrudan değişken tanımı yapılmasına izin verilmez.

F# ağırlıklı olarak fonksiyonel (bildirimsel) bir dil olmakla birlikte yordamsal yapıları da destekler. Bir sonraki örneğimizde F# ile nasıl prosedürel kod yazıldığını görüyorsunuz

```
(* 01_2_08.1.fsx *)
(* Yordamsal (fonksiyonel olmayan) yaklaşım *)

let liste = [1..10]

let mutable ikiyeBölünenler = []
let mutable ikiyeBölünmeyenler = []

for d in liste do
    if d % 2 = 0 then
        ikiyeBölünenler <- ikiyeBölünenler @ [d]
    else
        ikiyeBölünmeyenler <- ikiyeBölünmeyenler @ [d]
printfn "İkiye bölüneneler = %A" ikiyeBölünenler
printfn "İkiye bölünmeyenler = %A" ikiyeBölünmeyenler
```

Aşağıda F#'ın dil özellikleri ile uyumlu (idomatic) örnek kod parçasını görebilirsiniz.

```
(* 01_2_08.1.fsx *)
(* Bildirimsel (fonskyionel) yaklaşım *)
let liste = [1..10]
let ikiyeBolünebilirMi x = x % 2 = 0

let ikiyeBölünenler = liste |> List.filter ikiyeBolünebilirMi
printfn "İkiye bölüneneler = %A" ikiyeBölünenler

let ikiyeBölünmeyenler = liste |> List.filter (ikiyeBolünebilirMi >> not)
printfn "İkiye bölünmeyenler = %A" ikiyeBölünmeyenler
```

Toparlamak gerekirse;

- İki yaklaşımın kodlama stilleri birbirinden farklıdır. Yordamsal dillerde yapılacak her işlem adım adım tarfi edilir ve sonuç olarak yazılması gereken kod miktarı da fazla olur. Örneklerimizdekine benzer basit programlarda bile fonksiyonel yaklaşım ile %40 (10 satıra karşılık 6 satır) daha az kod yazılması mümkündür.
- Yordamsal dillerde çalıştırılan adımlar sonrasında varılan durumun takip edilmesi için değişkenler kullanılır. Bu değişkenlerin değerleri istenirse akış içinde herhangi bir aşamada değiştirilebilir. Ancak, fonksiyonel dillerde değişken kavramı yoktur bunun yerine değer ifadeleri (value expression) kullanılır. Değer ifadelerin değerleri ilk atandıkları andan sonra değiştirilemez.
- Çalıştırma sırası yordamsal dillerde önemlidir, çünkü durum takibi değişkenler ile yapılır ve her adım çalıştırıldıktan sonra bu değişkenlerin değeri değişebilir. Bu nedenle yordamsal dillerde kodun değerleme sırası önemlidir. Ancak, fonksiyonel dillerde ifadelerin değerleri atandıktan sonra değiştirilemediği için ve fonksiyonel programlar durumsuz oldukları için değerleme sırası önemli değildir.

- Fonksiyonel dillerde fonksiyonlar birinci sınıf vatandaştırlar. Bir fonksiyon başka bir fonksiyonu girdi parametresi olarak alıp çıktı olarak döndürebilir. Yordamsal dillerin bir kısmında da bu mümkündür. Ancak, yordamsal dillerde fonksiyonları girdi ve/veya çıktı olarak kullanmak daha fazla kod yazılmasını ve hata kontrollerinin dikkatli yapılmasını gerektirir.
- Yordamsal dillerde akış kontrolü için döngü (for/while), koşullu dallanma (if/else, switch) ve fonksiyon tanımları kullanılır. Fonksiyonel dillerde ise akış kontrolü için sadece öz yinelemeli (recursive) fonksiyonların kullanılması yeterlidi. Fonksiyonel dillerde akış kontrolü en alt seviyede derleyici tarfından otomatik oluşturulur.
- Yordamsal dillerde kullanılan temel veri yapıları değişkenler ve diziler (array) gibi içeriği değiştirilebilen yapılarıdır. Fonksiyonel dillerde ise yığım (collection) kullanımı ön plandadır.

Bilgi Kutusu (BİLGİ): Dizilerin(array) kapasitesi sabittir ve değiştirilemez. Koleksiyonların (collection) kapasitesi ise fiziksel kapasitenin izin verdiği sınırlara kadar büyüyebilir. Diziler ve koleksiyonlar hem yordamsal hem de fonksiyonel dillerde kullanılır. Ancak, fonksiyonel dillerde koleksiyon kullanımı tavsiye edilen en iyi pratikler arasında sayılır.

Yordamsal diller bir çok sektörde kullanılan ana akım dillerdir, bu nedenle fonksiyonel dillere oranla popülerliği ve üretilen kod miktarı daha fazladır. Ancak, bulut tabanlı sistemlerin ve büyük veri odaklı veri işleme uygulamalarının popüler hale gelmesi ile birlikte F#, Clojure ve Haskell gibi fonksiyonel dillerin popülerliği ve kullanımı artmaktadır. İfadelerini değerlerinin atandıktan sonra değiştirilememesi(immutability) ve fonksiyonların prensip olarak yan etkisinin (side effect) olmaması gibi temel yapısal özellikler bu dillerin paralel ve eş zamanlı işleme kabiliyeti gerektiren büyük veri projelerinde ön plana çıkmasını sağlar.

Sizler de bulut tabanlı büyük veri işleme uygulamaları veya benzer uygulamalar geliştirmek istiyorsanız F# veya farklı bir fonksiyonel programlama dilini öğrenerek kariyerinize pozitif bir katkı yapabilir, farklı mücadele ve fırsatlara açılan kapıları aralayabilirsiniz.

Bilgi Kutusu (NOT): Nesne yönelimli (object oriented) diller, yordamsal (imperative) ve bildirimsel (declarative) dillerden daha popüler olan üçüncü yaklaşımı temsil etmektedir.

2.Bölüm : F# Geliştirme Platformu

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.

2.1 Derleyici ve Yorumlayıcı Kavramları

Herhangi bir programlama dilinde geliştirme yapmak için ihtiyaç duyulan en basit bileşen **derleyicidir** (compiler). Derleyici metin dosyası olarak yazılan kodun dil kurallarına göre denetlenmesi ve optimize edilmesinden sonra çalıştırılabilir programın üretilmesini sağlar. Bazı programlama dilleri derleyici yerine **yorumlayıcı** adı verilen bileşeni kullanır. Yorumlayıcı derleyiciden farklı olarak kodun çalışma anında yorumlanarak çalıştırılmasını sağlar.

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

Gelin şimdi işletim sisteminin ne olduğundan bağımsız olarak bu iki bileşeni nasıl kullanacağımızı görelim.

2.2 FSC - F# Derleyicisi (F# Compiler)

F# derleyicisi açık kaynak kodlu bir bileşendir. F# derleyicisini Windows, Linux ve OSX 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
```

Örnek programımız çalıştığında konsol ekranına "Merhaba Dünya!" yazar. Konsol uygulamasına parametre geçilirse bu değerler de "Merhaba, " şeklinde sırasıyla konsol ekranına basılır ve uygulamanın sonlanması için kullanıcıdan ENTER tuşuna basması istenir.

Derleme için kod dosyasının bulunduğu klasöre konumlanıp komut satırına aşağıdaki komut çalıştırılır

```
$ fsc 02_1_01.fs -o merhaba.exe --target:exe
```

- **02_1_01.fs :** Derlenecek kodun bulunduğu dosya adı
- **-o merhaba.exe :** Derleme sonrası üretilecek dosyasının adı
- **--target:exe :** Derleme sonrası üretilecek dosyanın formatı, bizim örneğimizde exe

Komut çalıştırıldıktan sonra kod dosyasının bulunduğu klasör altında **merhaba.exe** isimli bir konsol uygulaması oluşturulur. Uygulamayı test etmek için konsol ekranına aşağıdaki komutu yazabilirsiniz.

```
$ merhaba.exe "Arda" "Ali"

# Uygulamanın çıktısı aşağıdaki gibi olacak

Merhaba Dünya!
Merhaba Arda.
Merhaba Ali.
-------
Sonlandırmak için lütfen ENTER'a basın.
```

Bilgi Kutusu (DİKKAT!): Linux veya OSX işletim sistemlerinde F# derleyicisini komut satırında **fsharpc** komutu ile çağırmalısınız.

F# derleyici parametrelerini Derleyici Seçenekleri sayfasından daha ayrıntılı olarak inceleyebilirsiniz.

2.3 FSI - F# Etkileşimli Ortamı

F# etkileşimli ortamı (FSI), kod yazım sürecindeki yaz-derle-dene döngüsünün daha hızlı bir şekilde yapılabilmesi için sunulan çok faydalı bir araçtır. FSI ile yazdığınız kodun seçtiğiniz kadarını derleme işlemi yapmadan çalıştırıp sonucunu hızlıca görebilirsiniz. FSI da F# derleyicisi gibi Windows, Linux ve OSX işletim sistemleri ile kullanılabilir. FSI'ı komut satırına **fsi** (Windows üzerinde) komutu ile çalıştırabilirsiniz.

Komut satırında **fsi** komutu çalıştırıldığında konsol ekranında FSI versiyon bilgisi gösterildikten sonra > simgesi belirecektir.

```
$ fsi
F# Interactive for F# 4.1
Freely distributed under the Apache 2.0 Open Source License

For help type #help;;
>
```

> simgesi FSI'in sizden kod girmenizi beklediğini ifade eder. FSI giriş modunda iken kodu **printfn "Merhaba Dünya!";;** yazıp ENTER tuşuna bastığınızda F# kodunuz derlenip çalıştırılarak konsol ekranında aşağıdaki çıktı üretilir.

```
Merhaba Dünya!
val it : unit = ()
```

Şimdi gelin FSI ile yapabileceklerimizin bir kısmını inceleyelim.

```
> 2 + 2;;
val it : int = 4

> let x = 42;;
val x : int = 42

> let y = 1;;
val y : int = 1

> let topla x y =
-     x + y
- ;;
val topla : x:int -> y:int -> int

> topla x y;;
val it : int = 43

> topla 11 12;;
val it : int = 23

> let x = -42;;
val x : int = -42

> topla x y;;
val it : int = -41
```

- **2 + 2;;** basit bir toplama işlemi
- **let x = 42;;** x isimli bir ifade tanımlayıp değerini 42 olarak belirle
- **let y = 1;;** y isimli bir ifade tanımlayıp değerini 1 olarak belirle
- **let topla x y =** topla isimli ve iki parametreli fonksiyon tanımına başla
- **x + y** topla isimli fonksiyonun kodunun ilk satırı
- **;;** topla isimli fonksiyon tanımını bitir
- **topla x y;;** daha önce tanımlanan x ve y değerleri ile topla fonksiyonunu çağır
- **topla 11 12;;** topla fonksiyonunu 11 ve 12 değerleri ile çağır
- **let x = -42;;** x ifadesini yeniden tanımla ve değerini -42 olarak belirle
- **topla x y;;** x ve y değerleri ile topla fonksiyonunu çağır

FSI çalıştırıldığı ilk andan itibaren (oturum) yapılan tüm değer ifadesi ve fonksiyon tanımlarını aklında tutar.

FSI'in girdiğiniz herhangi bir kodu yorumlamaya başlaması için kod satırını **;;** (çift noktalı virgül) ile bitirip ENTER tuşuna basmanız gerekir. Eğer **;;** kullanmadan ENTER satırına basarsanız FSI - koyarak bir sonraki satıra. -- simgesi FSI'in hala giriş modunda olduğunu ve kod girmenizi beklediği anlamına gelir. Örneğimizdeki **topla** fonksiyon tanımı FSI ile çalışırken birden fazla satırlı kodun nasıl yazılabileceğini gösterir.

FSI etkileşimli olarak kod girip deneme yapmanıza imkan vermenin yanı sıra F# script dosyalarınızı komut satırından çalıştırabilmenizi de sağlar.

```
(* 02_1_02.fsx *)

// Ekrana yaz
printfn "Merhaba Dünya!"

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

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

Yukarıdaki F# script dosyasını aşağıdaki komut satırı komutunu kullanarak çalıştırabilirsiniz.

```
$ fsi --exec 02_1_02.fsx -- Arda Ali
```

Bilgi Kutusu (NOT): Örnek kodda yer alan **fsi.CommandLineArgs** listesi scriptinize geçilen parametrelere kod içinden erişim imkanı sağlar.

FSI parametrelerini Etkileşimli Ortam Seçenekleri sayfasından daha ayrıntılı olarak inceleyebilirsiniz.

FSI'da scriptleri çalıştırırken aşağıdaki direktifleri kullanabilirsiniz.

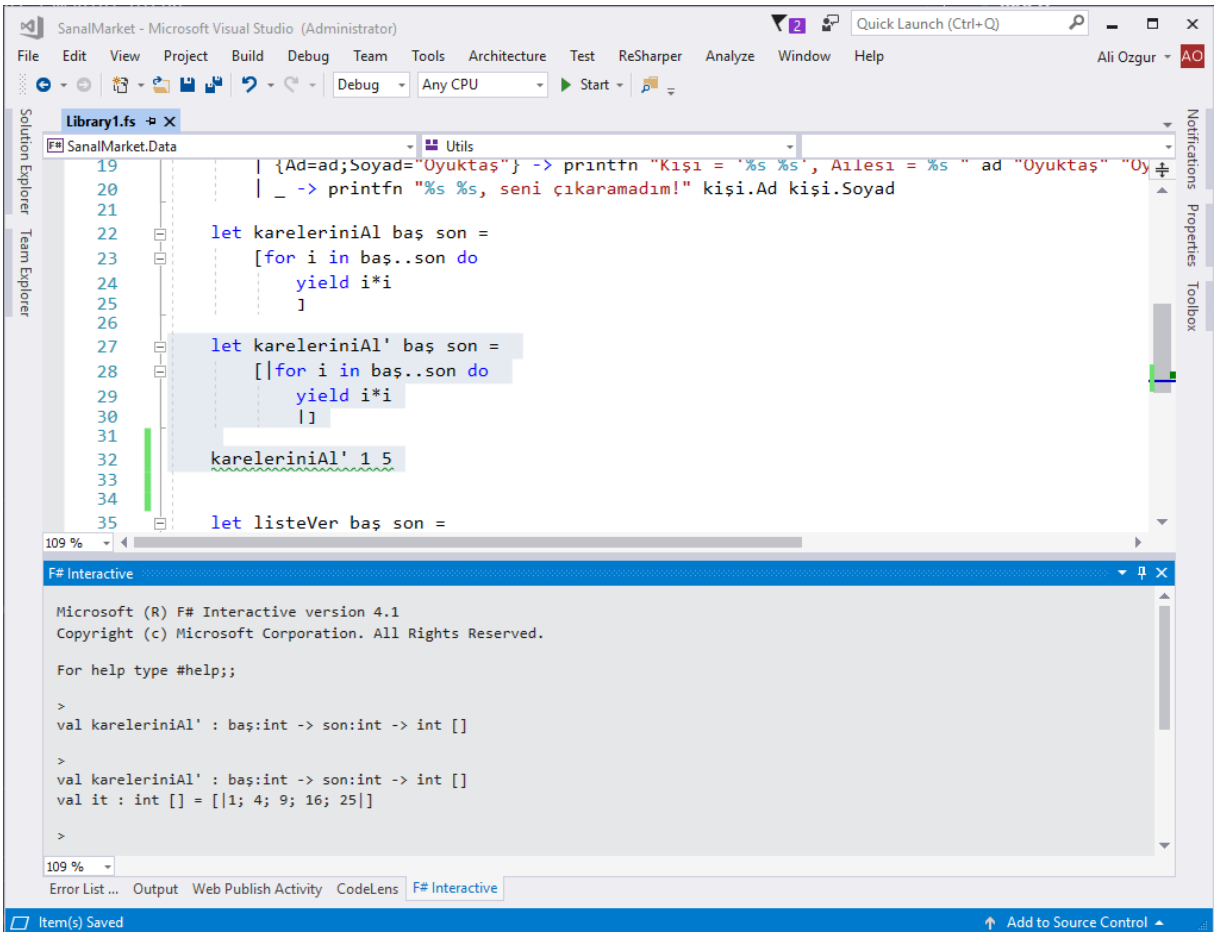
- **#help**, direktifler hakkında yardım almak için kullanılır.
- **#I**, Referans vermek istediğiniz kütüphane dosyalarının yer aldığı klasör yolu. Örneğin; #I "C:".
- **#load**, script dosyasını yükleyip çaıştırmak için kullanılır.
- **#quit**,FSI oturumunu bitirmek için kullanılır.
- **#r**, kütüphane dosyasına referans vermek için kullanılır.
- **#time ["on"|"off"]**, performans ve zamanlama bilgilerinin gösterilip gösterilmeyeceğini belirlemek için kullanılır. "on" ile aktif hale getirildiğinden FSI kodun çalışma zamanını, CPU'nun harcadığı zamanı ve çöp toplama (garbage collection) zamanını ölçer.

#I, #load, #r ve #time direktiflerini hem FSI giriş modunda iken hem de isterseniz script dosyalarınızın içinde kullanabilirsiniz.

```
(* 02_1_04.fsx *)

#time "on"

printfn "Ana script başladı"
#load "02_1_05.fsx"
printfn "Ana script devam edecek"

#I "libs//ali" // OSX, Linux
#I "libs\\ali" // Windows
```

```
#load "kare.fsx"

open Kare.Module1

// kare fonksiyonun grafiğini çiz
printfn "3'ün karesi = %f"  (kare 3.0)

#time "off"
printfn "Performans ve zamanlama ölçümü kapatıldı."
```

Bilgi Kutusu (İPUCU): Visual Studio, Visual Studio for Mac ve Visual Studio Code editörleri ile geliştirme yaparken seçilen satırları **Alt+Enter** tuş kombinasyonu ile FSI'ye gönderip sonuçları "F# Interactive" penceresinde görebilirsiniz. Bu editörlerin "F# Interactive" pencerelerinde aynı zamanda kendiniz de doğrudan komutlar yazarak deneme yapabilirsiniz



Bu bölümde 02_02.png nolu referans resim kullanılacak.

2.4 F# Standard Dosya Uzantıları

F# için aşağıdaki standard dosya uzantıları kullanılır.

- **fsx:** F# script dosyası. Bu dosyalar Visual Studio gibi geliştirme ortamları tarafından derleyiciye derleme için gönderilmez.
- **fs:** F# kod dosyası. Bu dosyalar Visual Studio gibi geliştirme ortamları tarafından derleyiciye derleme için gönderilir.
- **fsi:** F# kod (.fs) dosyasının içindeki yapıların imzalarını barındıran dosyası.

F# kod veya script dosyalarının yukarıdaki uzantılar ile oluşturulması zorunlu değildir. Bu uzantılar Visual Studio, Visual Studio Code ve Rider gibi F# destekleyen editörlerin kullandıkları varsayılan uzantılardır.

2.5 Derleyici ve Etkileşimli Ortam Değişkenleri

F# kod dosyaları içinde kodun etkileşimli ortamda (FSI) mı çalıştığı yoksa derleyici (FSC) tarafından mı derlenmekte olduğu **INTERACTIVE** ve **COMPILED** ortam değişkenlerinin değerleri incelenerek bilinebilir. FSI ortamında **INTERACTIVE** değişkenininin değeri **true** iken, FSC ile derleme yapılırken **COMPILED** değişkeninin değeri **true** olacaktır.

```
let topla x y =
#if INTERACTIVE
    // FSI ile çalıştırılırsa
    printfn "Etkileşimli"
    x + y
#endif
#if COMPILED
    // Derleyici tarafından derlenirse
    printfn "Derlenmiş"
    x + y
#endif

// TEST
topla 1 1
```

2.6 Geliştirme Araçları

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

Bu bölümde 02_01.png nolu referans resim kullanılacak.

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

Bilgi Kutusu (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

2.7 Merhaba F#

Örnek projemiz için aşağıdaki kurulumların yapılması gerekiyor.

- .NET Core
- Visual Studio Code

Visual Studio Code kurulumu tamamlayıp editörü açtıktan sonra **Ctrl+P** (OSX Cmd+P) kısayol tuş kombinasyonuna basıp **ext install Ionide-fsharp** komutu ile yardımcı bir araç olan Ionide'ın kurulumunu da yapmalısınız.

Adım-1: .NET Core kurulumunuzu kontrol etmek için komut satırı veya terminalde aşağıdaki komutu çalıştırın. Bu komut ekranına **2.0.0** benzeri bir versiyon değer yazdırmalı

```
$ dotnet --version
```

Adım-2: Komut satırında proje'yi kaydedeceğiniz klasöre konumlanıp aşağıdaki komutu çalıştırın. Bu komut yeni bir F# projesi oluşturur.

```
$ dotnet new console -lang F# -o Merhaba
```

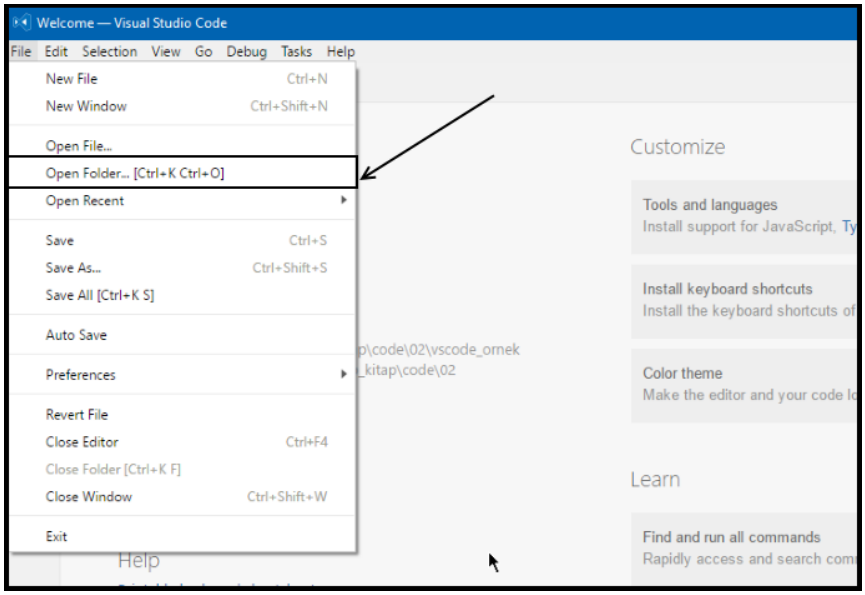
Komutun çalışması bittikten sonra proje klasörünün altında **Merhaba** isimli yeni bir klasör oluşturulur. **Merhaba** isimli klasörün altında aşağıdaki dosya ve klasörleri göreceksiniz.

- **Merhaba.fsproj**, proje tanım dosyası
- **Program.fs**, konsol programımızın kodunun yer alacağı dosya
- **obj**, F# derleyicisinin oluşturduğu ara klasör
- **bin**, derleme sonrası konsol uygulamasının yer alacağı klasör (bu klasör ilk derlemeden sonra oluşacak)

Adım-3: Aşağıdaki **build** komut ile projemizi derleyip sonra da **run** komutu ile konsol uygulamamızı çalıştırıalım

```
$ dotnet build

$ dotnet run
```

Adım-4: Gelin şimdi örnek uygulamamızda Visual Studio Code kullanarak kod değişikliklerimizi yapalım. Bunun için Visual Studio Code editörünü açıp **File -> Open Folder** menüsünden proje klasörünüzü (Merhaba) seçin



Bu bölümde 02_vscode_ornek/01.png nolu referans resim kullanılacak.

Adım-5: Proje klasörünüzü açtıktan sonra **Merhaba** klasörü altında **.vscode** isimli yeni bir klasör ve bunun altında da **launch.json** ve **tasks.json** isimli iki dosya ekleyin.

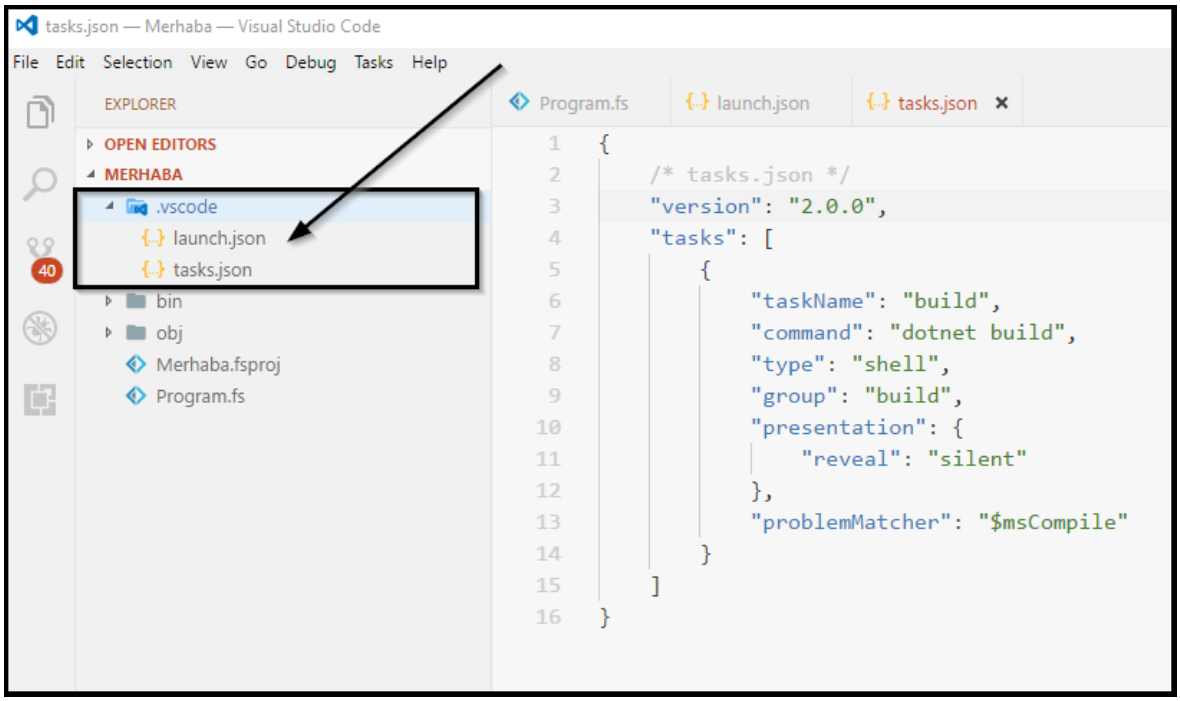
launch.json dosyasının içeriğini aşağıdaki gibi oluşturun.

```
{
    /* launch.json */
    "version": "0.2.0",
    "configurations": [
        {
            "name": ".NET Core Launch (console)",
            "type": "coreclr",
            "request": "launch",
            "preLaunchTask": "build",
            "program":
"${workspaceRoot}/bin/Debug/netcoreapp2.0/Merhaba.dll",
            "args": [],
            "cwd": "${workspaceRoot}",
            "stopAtEntry": false,
            "console": "internalConsole"
        }
    ]
}
```

Bu dosyadaki * **<insert-target-framework-here>** ibaresini silip yerine **netcoreapp2.0** yazın * **<insert-project-name-here>** ibaresini silip yerine **Merhaba** yazın

tasks.json dosyasının içeriğini de aşağıdaki gibi oluşturun

```
{
    /* tasks.json */
    "version": "2.0.0",
    "tasks": [
        {
            "taskName": "build",
            "command": "dotnet build",
            "type": "shell",
            "group": "build",
            "presentation": {
                "reveal": "silent"
            },
            "problemMatcher": "$msCompile"
        }
    ]
}
```



Bu bölümde 02_vscode_ornek/02.png nolu referans resim kullanılacak.

Adım-6: Visual Studio Code'da **F5** kısayol tuşuna basarak kodumuzu Debug modunda derleyip çalıştıralım. **DEBUG CONSOLE** panelinde **Hello world from F#** mesajını görebilirsiniz.

Bu bölümde 02_vscode_ornek/03.png nolu referans resim kullanılacak.

Adım-7: Konsol uygulaması projesine **Utils.fs** isimli yeni bir dosya ekleyip içeriğini de aşağıdaki gibi oluşturun

```
(* \code\02\vscode_ornek\Merhaba\Utils.fs *)
namespace Utils
    module Matematik =
        // Kare fonksiyonu
        let kare x = x * x
        //Küp fonksiyonu
        let küp x = (kare x) * x
```

Adım-8: Utils.fs dosyasını oluşturduktan sonra üstüne çift tıklayarak açın ve **F1** kısayol tuşuna basın. Açılan alanda **F#** yazarak gelen listeden **"Add Current File to Project"** seçeneğini seçin. Utils.fs dosyasını projeye ekledikten sonra **F1** kısayol tuşu ile açılan alana tekrar **F#** yazarak gelen listeden **"Move File Up"** seçeneğini seçin.

Adım-9: Program.fs dosyasının içeriğini aşağıaki gibi değiştirip **Ctrl+Shift+B** (OSX'de Cmd+Shift+B) ile konsol uygulamanızı derleyin.

```
(* \code\02\vscode_ornek\Merhaba\Program.fs *)

open System
open Utils

[<EntryPoint>]
let main argv =
    printfn "F# ile merhaba dünya"
    printfn "3'ün karesi %d" (Matematik.kare 3)
    printfn "3'ün küpü  %d" (Matematik.küp 3)

    0 // çıkış kodu
```

Adım-10: Komut satırına geçip **run** komutu ile uygulamayı çalıştırın

```
$ dotnet run
```

Ekran çıktısı aşağıdaki gibi olacaktır.

```
F# ile merhaba dünya!

C:\personalgit\fsharp_kitap\code\02\vscode_ornek\Merhaba>dotnet run
F# ile merhaba dünya
3'ün karesi 9
3'ün küpü  27
```

Adım-11: Konsol uygulamasının EXE dosyasını oluşturmak **Merhaba.fsproj** dosyasını açıp içeriğini aşağıdaki gibi değiştirmelisiniz.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <RuntimeIdentifier>win10-x64</RuntimeIdentifier>
    <RuntimeIdentifier>win7-x64</RuntimeIdentifier>
  </PropertyGroup>
  <ItemGroup>
    <Compile Include="Utils.fs" />
    <Compile Include="Program.fs" />
  </ItemGroup>
</Project>
```

Adım-12: Komut satırında **publish** komutunu çalıştırın.

```
$ dotnet publish -c Release
```

3.Bölüm: F# Temelleri

Bu bölümde önce F#'ın söz dizimi kurallarını inceliyoruz. Daha sonra basit (int,string,bool) ve temel veri tiplerini (değer grubu, unit, listeler, diziler) ele alıp F#'ın yapı taşları olan fonksiyonların ayrıntılarına bakıyoruz. Son olarak faydalı kod organizasyonu ipuçları ile bölümü tamamlıyoruz.

3.1 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 verilmeli. Girintileri oluşturmak için 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 çifti kullanımına gerek duyulmuyor. 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

// Modül tanımı
module Modül1 =
    // Aşağıdaki satırlar girinti verildiği için Modül1 alanına aittir
    let sayı' = 43
    let kırkÜçEkle x = sayı' + x

// Aşağıdaki satırda girinti yok o nedenle Modül1 ile aynı alana yani
Global alana ait
let sayı'' = 44
```

```
// Modül1 alan adı ekleyerek kırkÜçEkle fonksiyonunu Global kod alanından
kullanabiliriz
Modül1.kırkÜçEkle 44

// Global kod alanında fonksiyon tanımı
let birArttırVeKaresiniAl x =
    // Aşağıdaki satırlar girinti verildiği için birArttırVeKaresiniAl
alanına aittir
    let t = x + 1
    t * t

// sayı'' değeri birArttırVeKaresiniAl fonksiyonu ile aynı yani Global kod
alanında
birArttırVeKaresiniAl sayı''

// Global kod alanında fonksiyon tanımı
let çiftMiTekMi x =
    // Fonksiyonun kod alanı içinde tanımlı kod
    if x % 2 = 0 then
        // If koşulu kod alanı
        true
    else
        // Else koşulu kod alanı
        false


// 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 kod alanı
            "Çift"
        | _ ->
            // Koşul kod alanı
            "Tek"
```

```
// Müdül2 kod alanındaki çiftMiTekMi fonksiyon çağırısı
Modül2.çiftMiTekMi 12

// Global kod alanındaki çiftMiTekMi fonksiyon çağırısı
çiftMiTekMi 12
```

F#'da modül alan adları **ModüleAdı.** şeklinde kullanılarak modül içindeki değerlere ve fonksiyonlara erişilebilir. Aynı kod alanına ait ifadeler kendi yerel kod alanlarından bir üst seviyedeki (dışındaki) kod alanlarından ifadeleri kullanabilir.

```
(* 03_1_01.1fsx *)
// Global alanda tanımlı değer
let kırkİki = 42

// Global alanda tanımlı fonksiyon
let kırkİkiEkle x =
    // Global alandaki kırkİki değerii fonksiyon içinden kullanabiliriz
    kırkİki + x

// Modül tanımı
module Modül1 =
    // bir değeri Modül1 kod alanında
    let bir = 1


// Modül1 alan adında yer alan bir değerine Modül1.bir şeklinde
erişebiliriz
kırkİkiEkle Modül1.bir
```

"let" Anahtar Kelimesi

F# fonksiyonel bir dil olduğu için **değişken**, **değişken tanımlama** ve **değişkenin değerini değiştirme** gibi kavramlar kullanılmaz. Tüm fonksiyonel dillerde olduğu gibi F#'ın dil yapısı **ifade** (expression) denilen kavram üzerine inşaa edilmiştir. İfadelerin değerleri değişkenlerde olduğu gibi program akışı sırasında değiştirilemez.

"**let**" anahtar kelimesi F#'da isimlendirilmiş değer ifadelerinin ve fonksiyonların (ki onlar da birer ifadedir) tanımlanması için kullanılır. Genel yapısı şöyledir

```
// Basit değer ifadesi (tek satır)
let değerAdı = değer

// Basit değer ifadesi (çoklu satır)
let değerAdı =
    değer

// Fonksiyon (tek satır)
let fonksiyonAdı girdi1 .... girdiN = fonksiyon kodu

// Fonksiyon (çoklu satır)
let fonksiyonAdı girdi1 .... girdiN =
    fonksiyon kodu
```

Şimdi gelin yukarıdaki kurallara göre **let** kullanarak bazı değer ifadeleri tanımlayalım

```
(* 03_1_01.fsx *)

// Basit değer ifadesi tanımlama
let sayı = 12
let metin = "F# ile fonksiyonel programlama"
let pi = 3.14
let cevap = true

// Tek satırda birden fazla değer ifadesi tanımlama
let a,b,c = 1,2,3

// Daha karmaşık değer grubu tipinden değerler de tanımlanabilir
let x,y,z = (42,"F# ile Fonksiyonel Programlama", 3.14)

// Fonksiyon tanımlama
let küp x = x * x * x

// Öz yinelemeli fonksiyon tanımlama
let rec fib n = if n <= 1 then n else fib(n - 1) + fib(n - 2)
```

"let" anahtar kelimesi ifadelere değerlerinin bağlanmasını (binding) sağlar, bu nedenle diğer dillerdeki gibi klasik anlamda bir atama imkanı sağlamaz. Basit değer ifadelerine bağlanan değer genelde int, string, bool gibi basit veri tipleridir. Fonksiyon ifadeleri için ise bağlanan değerler fonksiyonun kodunu oluşturan alt ifadelerdir.

"let" ile tanımlanan basit veya fonksiyon ifadelerinin mutlaka ama mutlaka bir değer olmalıdır.

```
// Hata! Herhangi bir değer bağlanmamış
let sayı

//Doğru
let sayı = 42

// Hata! Herhangi bir fonksiyon ifadesi bağlanmamış
let fonksiyon girdi

// Doğru
let fonksiyon girdi = girdi + 1
```

"let" anahtar kelimesi modül seviyesinde, sınıf seviyesinde veya fonksiyon tanımı içinde kullanılabilir. İfadeler, tanımlandıkları satırdan sonra aynı kod alanı (scope) içinden (modül, sınıf veya fonksiyon tanımı içinden) veya alt kod alanları içinden kullanılabilir.

```
(* 03_1_02.fsx *)

// Global alanda (Program) let ile değer tanımlama
let globalSayı = 42

// Hata! kare fonksiyonu henüz tanımlanmadı
// kare globalSayı

// Global alanda (Program) let ile fonksiyon tanımlama
let kare x = x * x

// Modül tanımı
module Modül1 =
    // Modül alanı içinde basit değer tanımlama
    let modülSayısı = 43

    // Modül alanı içinde fonksiyon tanımlama
    let kök x = (kare x) * x
```

```
// Fonksiyon alanında yerel değer ifadeleri tanımlama
let yerDeğiştir x y =
    let ix = y
    let iy = x
    (ix,iy) // Değer grubu tipinden fonksiyon çıktısı


(* --- Kurgumuzu test edelim --- *)

kare globalSayı

// modülSayısı global alandan erişilebilir değil
//kare modülSayısı

// modülSayısı değerine Modül1 alan adı eklenerek erişebiliriz
kare Modül1.modülSayısı

// Modül1 içindeki kök fonksiyonuna Modül1 alan adı eklenerek global
alandan erişebiliriz
Modül1.kök 12

// yerDeğiştir fonksiyonu çağırısı
yerDeğiştir 1 2

// Hata! yerDeğiştir yerel alanında tanımlı ix ve iy sadece fonksiyon
içinde erişelebilir
//let tx, ty = ix,iy
```

"do" anahtar kelimesi

"do" anahtar kelimesi kullanılarak değer ifadesi veya fonksiyon tanımı olmasına ihtiyaç duyulmadan kod çalıştırılabilir. Program başlangıcında, modül tanımı başında veya sınıf tanımı içinde fonksiyon çağırısı yapılmadan otomatik çalışması istenen kod blokları "do" anahtar kelimesi kullanılarak çalıştırılabilir.

```
(* 03_1_03.fsx *)

do printfn "Program çalışmaya başladı"

// .... Program kodunuz
let kare x = x * x
printfn "2'nin karesi = %d" (kare 2)


module Modül1 =
    printfn "Modül çalışmaya baladı"
    let kare x = x * x
    printfn "Modül çalışması tamamlandı"

do printfn "Program sonlandı"
```

"do" kullanımı opsiyoneldir. Örneğimizdeki Modül1 içinde **do printfn** şeklinde yazılan ifadelerinin başındaki "do" kaldırılsa bile **prinfn** çağırısı çalıştırılır.

"do" kullanımı ile ilgili en önemli kısıtlama "do" sonrasında yazılan ifadenin dönüş değerinin **unit** tipinden olması zorunluluğudur. **unit** tipi F#'da özel bir tiptir ve **hiç birşey** anlamına gelir. Dolayısıyla, **do** ile başlayan ifadeler bir sonuç üretmemelidir.

Bilgi Kutusu (BİLGİ): unit tipini C,C++,Java ve C# dillerindeki **void** tipine benzediğini varsayabilirsiniz.

```
(* 03_1_04.fsx *)

// Hatalı kullanım
// 1 + 1 ifadesinin sonucu tam sayı tipinde ve 2
do 1 + 1

// Doğru kullanım.
// 1+1 sonucu olan 2 değeri ignore fonksiyonuna iletilir ve ignore unit
tipinde çıktı verir
do (1 + 1) |> ignore
```

Bilgi Kutusu (BİLGİ): ignore F# standard kütüphanesi ile gelen bir fonksiyondur. Tek bir giriş parametresi alır ve bu parametrenin tipi ne olursa olsun her zaman **unit** tipinden bir çıktı üretir.

Yorum Satırları

F#'da tek satırlı veya çok satırlı yorumlar kodun içine iki şekilde eklenir

- Tek satırlık yorumlarınız için // karakterlerini kullanabilirsiniz
- Birden fazla satırlık yorumlarınız için ise (* *) çiftini kullanabilirsiniz

// karakterleri sonrasında ve **(* *)** arasında yer alan ifadeler F# derleyicisi tarafından derlenmez ve dolayısıyla programınızın bir parçası olarak çalıştırılmaz

```
(* 03_1_05.fsx *)

// Tek satırlık yorum
// let x = 12

(*
    Çok satırlı
    yorum
*)

(*
    let kare x
        x * x
*)
```

Koşullu Derleme

Koşullu derleme, özellikle birden fazla platformu destekleyen veya versiyonlama anlamında geriye dönük uyumluluğun sağlanması ihtiyacı olan programlar için kullanılan bir yöntemdir.

Örneğin;

- Aynı uygulamanın mobil ve masaüstü versiyonlarının tek bir kod havuzunda geliştirilmesi,
- Tek bir uygulamanın aynı işletim sisteminin farklı versiyonlarında farklı işletim sistemi kütüphaneleri kullanması,

gibi durumlarda derleyiciye ipucu vermek için kod içinde koşullu derleme yapıları kullanılır. F#'da koşullu derleme için **#if #else #endif** derleyici makroları kullanılır.

```
(* 03_1_06.fsx *)
//------ ÖRNEK 1 ------//
#if v1
// v1 koşulunda çalışması istenen kod parçası
let kare x = x * x
#else
// v1 koşulu haricinde çalışması istenen kod parçası
let kare x = sprintf "Kare %d" x
#endif
```

```
// v1 ortam değişkeni tanımlı olmadığı için çıktı "Kare 4" olacaktır
kare 2

//------ ÖRNEK 2 ------//

let osx = true
#if osx
// osx koşulunda çalışması istenen kod parçası
let ortam() = "OSX"

#else
// osx koşulu haricinde çalışması istenen kod parçası
let ortam() = "OSX DEĞİL"

#endif

// osx değeri tanımlı ancak yine de çıktı "OSX DEĞİL" olacaktır
// Ortam değişkenlerini kodunuz içinde tanımlayamazsınız!
ortam()
```

Koşullu derleme ifadelerinin sınaması program derleme anında **ortam değişkenleri** kullanılarak yapılır ve değişken değeri derleyiciye parametre olarak sağlanır. Koşullu derleme kod çalıştığı sırada gerçekleşmez. Bu nedenle, ortam değişkenlerinin adı ile kod içinde tanımlanan sıradan değişkenler kullanılarak çalışma anında koşul sınaması yapılamaz.

F# derleyicisi ve F# interaktif için ortam değişkenlerini **--define** seçeneği ile aşağıdaki gibi tanımlayabilirsiniz.

- **fsharpc --define v1**
- **fsharpi --define osx**

Tanımlayıcı ve Anahtar Kelimeler

Değer ifadeleri tanımlarken kullandığımız ifade isimlerini **tanımlayıcılar**, F#'in dili içinde tanımlı özel tanımlayıcılara da **anahtar kelimeler** diyoruz.

```
(* 03_1_07.fsx *)

// sayı bir tanımlayıcı
// let ise anahtar bir kelime
let sayı = 42

let Al_i = 12
```

F#'da anahtar kelimeler dışındaki tanımlayıcıları oluştururken aşağıdaki kurallara uyulmalıdır

- Sadece herhangi bir **harf** veya **_** ile başlayabilir.
- **0 - 9 arasında** sayısal karakterler ile başlayamaz.
- Harfler, sayılar, alt çizgi (_) , tek tırnak (') karakterleri içerebilir.
- **Boşluk** (whitespace) veya tire (-) karakterini içeremez. Bu karakterler kullanılmak istenirse tanımlayıcı `` `` (iki ters kesme çifti) arasında yazılır.
- `` `` oluşturulan tanımlayıcı adında **TAB**, **satır başı** (New Line) veya `` (çift ters tırnak) karakterleri kullanılmaz.
- Tip isimleri, bileşim etiketleri (discriminated union), modül isimleri veya kod alanı (namespace) isimlerinde **'.'**, **'+'**, **'$'**, **'&'**, **'[', ']'**, '/', '\', '', '"', **'`' (ters tek tırnak)** karakterleri kullanılamaz.
- F# anahtar kelimeleri `` `` çifti içinde tanımlayıcı olarak kullanılabilir, bunun dışında anahtar kelimeler tanımlayıcı olarak kullanılamaz.

```
(* 03_1_08.fsx *)

// Doğru kullanım
let sayı = 42
//let -sayı = 42 // Hatalı

let _sayı = 42
//let 42sayısı = 42 // Hatalı

let mucize_sayı = 42
//let mucize-sayı = 42 // Hatalı

let kare x = x * x
//let -kare x = x * x // Hatalı

let _kare x = x * x
//let 42çarpıKare x = 42 * (x * x) // Hatalı

let kare_fonk x = x * x
//let kare-alma x = x * x // Hatalı

// Anahtar kelimenin tanımlayıcı olarak kullanımı
let ``let``= "Let ifadesi"
//let let = "Let ifadesi" // Hatalı

// Boşluklu tanımlayıcı ismi
let ``iki ile topla`` x = x + 2
//let iki ile topla x = x + 2 // Hatalı

//UTF-8 karakterlerin kullanımı
let çığırAçanSayı = 42
let π = 3.14
let cliché = "Klişe"

// f fonksiyonu
let f (x:float) = 2.0 * x + 4.0

// f' fonksiyonu, f fonksiyonun tersi
let f' (x:float) = 0.5 * x - 2.0
```

Bilgi Kutusu (DİKKAT!): F# derleyicisi kod dosyalarının karakter kodlamasının (encoding) UTF-8 olduğunu varsayar.

Anahtar Kelimeler (F# 4.1 itibariyle)

abstract and as assert base begin class default delegate do done downcast downto elif else end exception extern false finally for fun function global if in inherit inline interface internal lazy let match member module mutable namespace new null of open or override private public rec return sig static struct then to true try type upcast use val void when while with yield

Rezerve Edilmiş Anahtar Kelimeler

Gelecekte kullanılmak üzere aşağıdaki anahtar kelimeler F# tarafından rezerve edilmiştir. Bu anahtar kelimeler de ifadelerde tanımlayıcı olarak kullanılamaz

atomic break checked component const constraint constructor continue eager fixed fori functor include measure method mixin object parallel params process protected pure recursive sealed tailcall trait virtual volatile

Shebang

Kod veya script dosyalarının başında **#!** ile başlayan ve **shebang** (okunuşu şibenk) olarak adlandırılan özel bir karakter kombinasyonu kullanılabilir. Shebang ifadesi ile script dosyasındaki kodu çalıştırması istenen yorumlayıcı konumu tanımlanır. F# derleyicisi shebang satırlarını derlemez, bu satır Unix tabanlı işletim sistemlerinde **shell** tarafından yorumlanır.

Örneğin F# script dosyanızın başına aşağıdaki shebang komutunu eklerseniz Unix, Linux ve OSX işletim sistemlerinde dosyanızı komut satırına yazar yazmaz F# yorumlayıcısı **fsharpi** dosyanızın içindeki kodu çalıştıracaktır.

```
#!/bin/usr/env fsharpi --exec

(* 03_1_09.fsx *)
printfn "Merhaba Dünya!"
```

```
# Komut satırı
$ 03_1_09.fsx
```

3.2 Basit Veri Tipleri

Tüm programlama dillerinde herhangi bir verinin mutlaka bir tipi vardır. Sayı, metin, karakter ve evet/hayır şeklinde değer barındıran tiplere basit tipler denir. Programlama dilleri tasarımında tipler daha çok kavramsal büyüklükler olarak ele alınır ve asıl amaçları programlarımızdaki hataları derleme anında veya çalışma anında engellemektir. Tipler, program verisinin program akışı sırasında doğru kullanılmasını ve fonksiyonlar arasında veri aktarımının güvenli yapılmasını sağlar. Özetle; tipler ile ilgili tüm kaygı kavramsal seviyede veri dönüşümünün tutarlılığına odaklanmıştır.

F#'da basit tipler olarak adlandırdığımız 16 veri tipi vardır. F# bir .NET dili olduğu için tiplerden 15 tanesi doğrudan .NET tip sistemi tarafından tanımlanır.

F# tipi	Açıklama	.NET Tipi	Bellek Miktarı	Değer Aralığı	Örnek
sbyte	8-bit işaretli tam sayı	System.SByte	1 byte	-128 ile 127 aralığınd	42y, -11y
byte	8-bit işaretsiz tam sayı	System.Byte	1 byte	0 ile 255 aralığında	42uy, 200uy
int16	16-bit işaretli tam sayı	System.Int16	2 byte	-32768 ile 32767 aralığında	42s, -11s
uint16	16-bit işaretsiz tam sayı	System.UInt16	2 byte	0 ile 65,535 aralığında	42us, 200us
int/int32	32-bit işaretli tam sayı	System.Int32	4 byte	-2,147,483,648 ile 2,147,483,647 aralığında	42, -11
uint32	32-bit işaretsiz tam sayı	System.UInt32	4 byte	0 ile 4,294,967,295 aralığında	42u, 200u
int64	64-bit işaretli tam sayı	System.Int64	8 byte	-9,223,372,036,854,775,808 ile 9,223,372,036,854,775,807 aralığında	42L, -11L
uint64	64-bit işaretsiz tam sayı	System.UInt64	8 byte	0 ile 18,446,744,073,709,551,615 aralığında	42UL, 200UL
bigint	Rastgele uzunlukta tam sayı	System.Numerics.BigInteger	En az 4 byte	Herhangi bir tamsayı	42I, 1499999999999999999999999I
float32	32-bit işaretli ondalık sayı (ondalık kısım 7 basamağa kadar)	System.Single	4 byte	±1.5e-45 ile ±3.4e38 aralığında	42.0F, -11.0F
float	64-bit işaretli ondalık sayı (ondalık kısım 15-16 basamağa kadar)	System.Double	8 byte	±5.0e-324 ile ±1.7e308 aralığında	42.0, -11.0
decimal	128-bit işaretli ondalık sayı (ondalık kısım 28-29 basamağa kadar)	System.Decimal	16 byte	±1.0e-28 ile ±7.9e28 arasında	42.0M, -11.0M
BigRational	Rastegele uzunlukta rasyonel sayı. Kullanmak için FSharp.PowerPack.dll referans verilmesli	Microsoft.FSharp.Math.BigRational	En az 4 byte	Herhangi bir rasyonel sayı.	42N, -11N
char	Unicode karakter	System.Char	2 byte	U+0000 ile U+ffff arasında	'x','\t'
string	Unicode metin	System.String	20 + (2 * metnin uzunluğu) byte	0'den 2 miliyar adetkaraktere kadar	"Merhaba Dünya!", "F# ile fonksiyonel programlama"
bool	Lojik 1 ve 0	System.Boolean	1 byte	Sadece iki olası değer, true veya false	true,false

Bu bölümde 03_01.png nolu referans resim kullanılacak.
Bu bölümde 03_02.png nolu referans resim kullanılacak.

F#'da "let" ile basit değer ifadesi tanımlama formatı şöyledir

```
let <değer adı>:<değer tipi> = <değer>
```

```
let sayı:int = 42
let metin:string = "42"
```

Değer ifadelerinde tip kullanımı opsiyoneldir. Yukarıdaki ifadeler aşağıdaki gibi de yazılabilir, bu durumda F# **tip çıkarsama** (type inference) ile ifadeye verilen değerin tipini otomatik olarak çıkarsar.

```
let sayı  = 42 // sayı değer ifadesinin tipi int olarak çıkarsanır

let metin = "42" // metin değer ifadesini tipi string olarak çıkarsanır
```

Fonksiyon tanımlarında girdi parametrelerinin ve dönüş değerinin tipi aşağıdaki şablona göre yapılır.

```
let <fonksiyon adı> (girdi1:<girdi1 tipi>) (girdi2: <girdi2 tipi>): <sonuç
tipi> = <fonksiyon kodu>
```

```
let topla (x:int) (y:int): string =
    sprintf "%d + %d = %d" x y (x+y)

topla 42 0
```

Fonskiyon girdi parametrelerinin veya dönüş değerinin tipinin tanımlanması opsiyoneldir. Tipler kullanılmadan yukarıdaki örnek aşağıdaki gibi de yazılabilir, bu durumda F# **tip çıkarsama** ile doğru tipleri çıkarsar.

```
let topla x y =
    sprintf "%d + %d = %d" x y (x+y)

let topla' (x:int) y =
    sprintf "%d + %d = %d" x y (x+y)

let topla'' x (y:int) =
    sprintf "%d + %d = %d" x y (x+y)

let topla''' x y : string =
    sprintf "%d + %d = %d" x y (x+y)

topla 42 0
topla' 42 0
topla'' 42 0
topla'' 42 0
```

Fonksiyon girdi paremetrelerinin tanımında tipleri de tanımlamak isterseniz parametreleri **çift parantez ()** içine almalısınız. Çift parantez kullanmadan tipleri tanımlarsanız F# derleyicisi tanımınızı farklı bir şekilde yorumlayabilir.

Bilgi Kutusu (İPUCU): Basit değer ve fonksiyon tanımlarında tipleri kullanmayarak daha okunabilir kod yazılabilir.

F#'da sayısal değer alan ifadeler 2'lik (binary), 8'lik (octal) ve 16'lık (hexadecimal) sayı tabanlarını kullanılarak da tanımlanabilir.

```
// 2'lik (binary) ifade formatı
let değer_ifades = 0b[0 veya 1]

// 8'lik (ocatl) ifade formatı
let değer_ifades = 0o[0..7]

// 16'lık (hexadecimal) ifade formatı
let değer_ifadesi_adı = 0x[0..1 A..F]
(* 03_2_02.fsx *)

// 2'lik (binary) olarak 29
let ikilik = 0b11101

// 8'lik (ocatl) olarak 29
let değer_ifades = 0o35

// 16'lık (hexadecimal) olarak 29
let değer_ifadesi_adı = 0x1D
```

Artimetik İşlemler

F#'da yer alan 16 basit veri tipinden 9'u sayısal değerleri tarif etmek için kullanılan tiplerdir. Bu sayısal veri tiplerini ve aşağıdaki artimetik operatörleri kullanarak F# ile aritmetik işlemler yapabilirsiniz.

Operatör	Açıklama	Örnek	Sonuç
+	Toplama	1 + 2	3
-	Çıkarma	2-1	1
*	Çarpma	3 * 4	12
/	Bölme	4 / 2	2
**	Kare	2.0 ** 3.0	8
%	Mod	4 % 3	1

Bu bölümde 03_03.png nolu referans resim kullanılacak.

Artimetik operatörler ile işlem yapılırken değer aşımı durumu F# tarafından kontrol edilmez, bu nedenle herhangi bir hata almazsınız. Değer aşımı durumunu şöyle tanımlayabiliriz; örneğin 127y değerine sahip 8-bit işaretli bir tam sayıya 1y eklediğinizde 8-bit işaretli tam sayılar aralığında pozitif üst limit 127y olduğu için sonuç 128y olamaz. Üst limit aşımında sonuç negatif olacak alt limit aşımında ise sonuç pozitif olacaktır. Değer aşımı sadece toplama ve çıkarma işlemleri için diğer artimetik işlemler için de geçerlidir.

Bu durum sayıların 2'li sayı sistemindeki ifade şeklie ve 2'li sayı sistemi aritmetiğinin doğal sonucudur. Şöyle ki; 8-bit işaretli tam sayılar 2'li sayı sisteminde 8 bit ile temsil edilirler. Ancak bu 8 bit'den en soldaki 1. bit işaret bitidir. Pozitif sayılar için bu işaret bitinin değeri 0, negatif sayılar için de 1 olmalıdır. Buna göre

- 127y = 01111111, soldan ilk bit 0
- -128y = 10000000, soldan ilk bit 1

Aşağıda bu sonucun nasıl oluştuğu basit artimetik adımları şeklinde verilmektedir

2'li sayı sistemi aritmetiğinde 1 + 1 = 2 -> sonuç 0 elde var 1
10'lu sayı sistemi aritmetiğinde 1 + 9 = 10 -> sonuç 0 elde var 1

Pozitif yönde aşım

```
 127y = 01111111
 1y   = 00000001
+ ----------------
        10000000   (-128)
```

Negatif yönde aşım
-128y = 10000000
-1y = 11111111
+ ----------
01111111 (127)

En soldaki 1 + 1 = 0 elde var 1 ancak temsil 8 bit ile yapıldığı için eldeyi solda aktarabileceğimiz basamak kalmaz, bu nedenle elde değeri göz ardı edilir

```
(* 03_2_03.fsx *)

// 8-bit işretli tam sayı -128 ile 127 aralığında değer alabilir
let sonuç1 = 127y + 1y // Sonuç -128y
let sonuç2 = -128y + (-1y) // Sonuç 127y


// 32 bit işaretli tam sayı -32768 ile 32767 aralığında değer alabilir
let sonuç3 = 32767s + 1s // Sonuç -32768s
let sonuç4 = -32768s + (-1s) // Sonuç 32767s

// Çarpma işleminde aşım
let sonuç5 = -128y * 3y // Sonuç -128y

// 2'li sayı düzeninde ifadeler ve aşım
let a = 0b01111111y // 127y
let b = 0b00000001y // 1y
let sonuç6 = a + b  // -128y

let a' = 0b10000000y // -128y
let b' = 0b11111111y // -1y
let sonuç7 = a' + b' // 127y
```

Toplama, çıkarma, çarpma, bölme, kare alma ve mod alma operatörlerine ilave olarak F# standard kütüphanesinde matematiksel işlemlerde kullanabileceğiniz aşağıdaki fonksiyonlar da yer alır

Fonksiyon	Açıklama	Örnek	Sonuç
abs	Sayının mutlak değerini alma	abs -42.0	42.0
ceil	Yukarı doğru en büyük tam sayıya yuvarlama	ceil 42.001	43.0
exp	e'nin kuvveti	exp 1	2.7183
floor	Aşağı doğru en küçük tam sayıya yuvarlama	floor 42.999	42.0
log	Doğal logaritma (ln). 10 tabanında log10	log 2.71828	1.0
sqrt	Karekök	sqrt 4.0	2.0
cos	kosinüs	cos 0.0	1
sin	Sinüs	sin 0.0	0
tan	Tanjant	tan 1	1.557
pown	Sayının n. kuvvetini alma	pown 2 3	8

Bu bölümde 03_04.png nolu referans resim kullanılacak.

Bilgi Kutusu (İPUCU): Daha gelişkin matemtiksel fonksiyonlara ihtiyacınız varsa .NET platformu için açık kaynaklı olarak geliştirilen ve F# ile de kullanabileceğiniz Math.NET kütüphanesine göz atabilirsiniz.

Tipler Arası Dönüşüm

F# güvenli tipli (safe type) bir dildir, bunun bir sonucu olarak

- Her değerin tipi doğrudan veya tip çıkarsama ile derleme anından bilinmelidir
- Tipler arasındaki dönüşümler açık açık belirtilmelidir

Diğer bazı dillerde olduğu gibi F# derleyicisi, formel olarak bazı koşullarda yapabilecek olsa bile, derleme anında basit veri tipleri arasında otomatik dönüşüm yapmaz. Örneğin 32 bit işaretli bir tam sayıyı girdi olarak alan bir fonksiyona 8 bit işaretli bir tam sayıyı girdi olarak doğrudan göndermezsiniz.

```
(* 03_2_04.fsx *)
let kare x = x * x
let sayı = 2y

// Aşağıdaki kullanım hatalı
// F# tip çıkarsama mekanizması kare fonksiyonun girdi olarak 32 bit
// işaretli tam sayı beklediğini çıkarsadı
//let sonuç = kare sayı

// Doğru kullanım
let doğruSayı  = 2 // Tip çıkarsama doğruSayı değerinin tipini int olarak
çıkarsadı
let sonuç = kare doğruSayı

// Fonksiyon girdi parametresinin tipini doğrudan tanımlayarak alternatif
yaklaşım
let kare' (x:sbyte) = x * x
let sonuç' = kare' sayı

// 8 bit işaretli sayısı 32 bit işaretli sayıya çevirerek kullanım
let sonuç'' = kare (int sayı)


// 64 bit işaretli tam sayı
let büyükSayı = System.Int64.MaxValue - 1L // 9223372036854775806L

// 32 bit işaretli tam sayıya çevirmek istediğimizde değer aşımı meydana
gelir
let intSayı = int büyükSayı // -2
```

F# basit veri tipleri arasındaki dönüşüm işlemlerini sizin kodlamanızı bekler. Tip dönüşümü yaparken, özellikle sayısal tipler için, bölümün başındaki tabloda verilen değer aralıklarını kontrol etmelisiniz. Kaynak tip ile hedef tip aralıkları uyumlu değilse kaynak tipteki bir değer hedef tipteki geçerli aralığın dışında kalabilir. Değer uyumsuzluğu durumunda ise değer aşımı oluşur.

Aşağıdaki tabloda basit tipler arasındaki dönüşümler için kullanılan fonksiyonlara yer verilmiştir

Operatör	Açıklama
byte	8-bit işaretsiz byte'a dönüştür.
sbyte	8-bit işaretli byte'a dönüştür.
int16	16-bit işaretli tam sayıya dönüştür.
uint16	16-bit işaretsiz tam sayıya dönüştür.
int32, int	32-bit işaretli tam sayıya dönüştür.
uint32	32-bit işaretsiz tam sayıya dönüştür.
int64	64-bit işaretli tam sayıya dönüştür.
uint64	64-bit işaretsiz tam sayıya dönüştür.
nativeint	Platform için tanımlı işaretli tam sayı tipine dönüştür.
unativeint	Platform için tanımlı işaretsiz tam sayı tipine dönüştür.
float, double	64-bit IEEE standardına uygun ondalık sayıya dönüştür.
float32, single	32-bit IEEE standardına uygun ondalık sayıya dönüştür.
decimal	System.Decimal tipine dönüştür.
char	System.Char tipinden Unicode karaktere dönüştür.
enum	Numaralı liste'ye dönüştür

Bu bölümde 03_04_01.png nolu referans resim kullanılacak.

```
(* *3_2_4a.fsx *)

// işaretsiz byte
let sayı1= 42y

//işaretli byte
let sayı2 = byte sayı1

// 16-bit işaretli tam sayı
let sayı3 = int16 sayı1

// 16-bit işaretsiz tam sayı
let sayı4 = uint16 sayı1
```

```
// 32-bit işaretli tam sayı
let sayı5 = int sayı1

// 32-bit işaretli ondalık sayı
let sayı6 = float32 sayı1

// 64-bit işaretsiz ondalık sayı
let sayı7 = float sayı1

// Karakter sayısal koddan karakter elde etme
let char1 = char sayı1
```

Değer aşımlarının F# tarafından kontrol edilmesini ve aşım durumunda hata üretilmesini istiyorsanız F# standard kütüphanesinde yer alan **Checked** modülünü kullanmalısınız. Bu modülü kullanmak için kaynak kodu dosyanızın başında **open Checked** ifadesini yazmanız yeterlidir. Bu satırdan sonraki kod satırlarınız için F# **Checked** modülü içindeki artimetik operatör tip dönüşüm fonksiyonlarını kullanacaktır.

```
(* 03_2_4b.fsx *)

open System

// İşaretsiz 16-bit tam sayı maksimum değeri 65,535
let işaretsiz_16_bit = UInt16.MaxValue

// İşaretli 16-bit tam sayıya dönüştürelim
// Değer aşımı nedeni ile sonuç -1s olacaktır.
// Çünkü, işaretsi 65,535 değeri işaretli tam sayı aralığı dışında
let işaretli_16_bit = int16 işaretsiz_16_bit

// Değer aşımlarında hata üretilmesi için Checked modülüne referans
veriyoruz
open Checked

// Hata üretilir çünkü işaretsiz 65,535 değeri işaretli değer aralığının
dışında
let işaretli_16_bit_checked = int16 işaretsiz_16_bit
```

Karşılaştırma ve Eşitlik

Sayısal değerleri eşittir, eşit değildir, büyüktür, büyük eşittir,küçüktür ve küçük eşittir operatörleri ve **compare** standard kütüphane fonksiyonu ile karşılaştırabilirsiniz. Karşılaştırma operatörlerinin işlem sonucu her zaman **true** veya **false** mantıksal değerine eşittir. **compare** fonksiyonun dönüş değeri eşitlik durumunda 0, ilk girdi parametresi ikinciden küçük ise -1, ilk girdi parametresi ikinciden büyük ise 1 olur.

Operatör	Açıklama	Örnek	Sonuç
=	Eşittir	1 = 2	false
<>	Eşit değildir	1 <> 2	true
>=	Büyük eşittir	2 >= 2	true
>	Büyüktür	2 > 2	false
<=	Küçük eşittir	2 <= 2	true
<	Küçüktür	2 < 2	false
compare	Karşılaştır	compare 1 2	-1

Bu bölümde 03_07.png nolu referans resim kullanılacak.

```
(* 03_2_4c.fsx *)

// Operatörler ile karşılaştırma
let kırkİkiİleKarşılaştır x =
    if x > 42 then
        printfn " %d > 42" x
    else if x < 42 then
        printfn " %d < 42" x
    else
        printfn " %d = 42" x
```

```
// compare fonksiyonu ile karşılaştırma
let kırkİkiİleKarşılaştır' x =
    let sonuç = compare x 42

    if sonuç = 0 then
        printfn " %d = 42" x
    else if sonuç = 1 then
        printfn " %d > 42" x
    else
        printfn " %d < 42" x

kırkİkiİleKarşılaştır 43
kırkİkiİleKarşılaştır 41
kırkİkiİleKarşılaştır 42

kırkİkiİleKarşılaştır' 43
kırkİkiİleKarşılaştır' 41
kırkİkiİleKarşılaştır' 42
```

Bit Manipülasyonu

Sayısal veri tipinden değerleri bit seviyesinde ağıdaki operatörleri kullanarak değiştirebiliriz.

Operatör	Açıklama	Örnek	Sonuç
&&&	Lojik VE	0b1111 &&& 0b0011	0b0011
|||	Lojik VEYA	0xFF00 ||| 0x00FF	0xFFFF
^^^	XOR veya dışlamalı yada	0b0011 ^^^ 0b0101	0b0110
<<<	Sola kaydırma	0b0001 <<< 3	0b1000
>>>	Sağa kaydırma	0b1000 >>> 3	0b0001

Bu bölümde 03_05.png nolu referans resim kullanılacak.

```
(* 03_2_4d.fsx *)

// VE
let sonuç1 = 0b1111 &&& 0b0011

// VEYA
let sonuç2 = 0xFF00 ||| 0xFFFF
```

```
// XOR
let sonuç3 = 0b0011 ^^^ 0b0101

// SOLA KAYDIR
let sonuç4 = 0b0001 <<< 3


// SAĞA KAYDIR
let sonuç5 = 0b1000 >>> 3
```

Mantıksal/Lojik Değerler

F#'da mantıksal 1 ve 0 değerlerini tanımlamak için **bool** tipi kullanılır. Bool tipi **true** veya **false** şeklinde 1 bitlik iki değerden birini alabilir. Mantıksal **bool** tipindeki değerler ile VE, VEYA ve DEĞİL operatörleri kullanılarak **Bool Cebri** işlemleri yapılabilir.

Operatör	Açıklama	Örnek	Sonuç
&&	VE operatörü	true && false	false
||&	VEYA operatörü	true || false	true
not	DEĞİL operatörü	not false	true

Bu bölümde 03_06.png nolu referans resim kullanılacak.

```
(* 03_2_4e.fsx *)

// VE
let ikisiDeKırkİkidenBüyük x y = (x > 42) && (y > 42)

// VEYA
let enAzBiriKırkİkidenBüyük x y = (x > 42) || (y > 42)

// DEĞİL
let ikisiDeKırkikidenBüyükDeğil x y =
    not ( (x > 42) && (y > 42) )

ikisiDeKırkİkidenBüyük 43 44
ikisiDeKırkİkidenBüyük 42 43

enAzBiriKırkİkidenBüyük 40 43
enAzBiriKırkİkidenBüyük 40 41

ikisiDeKırkikidenBüyükDeğil 40 41
ikisiDeKırkikidenBüyükDeğil 43 44
```

Bilgi Kutusu (Bilgi)

Bir teoriye göre evrendeki tüm karmaşık sistemler sadece lojik VE, VEYA ve DEĞİL basit devreleri kombine edilerek oluşturulabilir.

Karakterler

F# karakter veri tipi desteği için .NET'in sağladığı imkanları kullanır. Karakterlerin tipi **char** olarak tanımlanır veya çıkarsanır. .NET'de karakterler 2 byte'lık unicode değerler olarak UTF-16 formatında ifade edilir. Basılabilir herhangi bir karakter tek tırnak çifti (' ') içinde yazılarak karakter değeri tanımlanır. Alternatif olarak tek tırnak çifti içine yazmak istediğiniz karakterin unicode kodunu yazarak da tanımlama yapılabilir.

Bilgi Kutusu (İPUCU): Karakterlerin unicode ifadeleri ve UTF-8, UTF-16 ve UTF-32 gibi kodlama yöntemleri kitabın kapsamı dışında olduğu için bu konudaki ayrıntılara girmiyoruz. Ancak, isterseniz unicode karakter kodlarını https://unicode-table.com adresinden inceleyebilirsiniz.

```
(* 03_2_05.fsx *)

let üHarfi = 'ü'
let sesliHarfler = ['a';'e';'ı';'i';'o';'ö';'u';'ü']

let üHarfiUnicode = '\u00FC'
let sesliHarflerUnicode =
['\u0061';'\u0065';'\u0131';'\u0069';'\u006F';'\u00F6';'\u0075';'\u00FC']
```

Alfabetik karakterler ilave olarak ASCII kod tablosunda kontrol karakteri olarak tanımlanan tab, yeni satır, satır başı gibi özel karakterler ile tek tırnak ('), çıft tırnak (") ve geri bölü () gibi F# dilinde özel anlamı olan karakterler de başlarına geri bölü () koyarak kullanılabilir.

```
(* 03_2_06.fsx *)
let tekTırnak = '\''
let çiftTırnak = '\"'
let geriBölü = '\\'
let tab = '\t'
let yeniSatır = '\n'
let satırBaşı = '\r'

printfn "tek tırnak %c, çift tırnak %c" tekTırnak çiftTırnak

// 'a' karakterinin sayısal unicode değeri
// int dönüşüm fonksiyonu kullanılarak elde edilir
let a = int 'a'

// 'a' karakterinin 8 bitlik işaretsiz sayı karşılığı
// karakter tanımının sonuna B koyarak elde edilir
let bitmap = 'a'B
```

Bilgi Kutusu (İPUCU): Bir karakterin sayısal karşılığını görmek için tip dönüşüm fonksiyonları kullanılabilir. Örneğin **let a = int 'a'** ifadesi ile "a" harfinin unicode kod tablosundaki sayısal karşılığı elde edilir. Ayrıca **let a = 'a'B** ifadesindeki gibi karakterin sonuna "B" tip tanımlayıcısını ekleyerek "a" harfinin 8 bit işaretsiz tam sayı karşılığı olan değeri bulabiliriz.

Metinler

F#'da metin değerlerini ifade etmek için çift tırnak çiftini (" ") kullanırız. Metin değerlerinin tipi **string** olarak ifade edilir. Çift tırnak çifti arasına yazılan tüm karakterler bir metin oluşturur. .NET ve F#'da metinler için UTF-16 unicode kodlama yöntemini kullanılır. Metin oluşturmak için tüm alfabetik karakterler ve kontrol karakterleri kullanılabilir.

```
(* 03_2_07 *)

// Çift tırnak ile metin tanımlama, unicode ve kontrol karakterleri

let metin1 = "F# ile fonksiyonel programlama"

let metin2 = "ali özg\u00FCr"
// Çıktı
// ali özgür

let metin3 = "\'Kitap Adı\' F# ile Fonksiyonel Programlama\n \"Yazar\" Ali
Özgür"
// Çıktı
(*
'Kitap Adı' F# ile Fonksiyonel Programlama
 "Yazar" Ali Özgür
*)
```

Çift tırnak çiftine ilave olarak F#'da çift tırnak üçlüsü çifti (""" """) de metin değerleri tanımlamak için kullanılabilir. Bu alternatif kullanım sayesinde metin değerinin içindeki çift tırnak (") ve tek tırnak (') karakterleri geri bölü kullanmadan yazılabilir.

```
// Çift tırnak üçlüsü ile metin değeri tanımlama
let metin4 = """ "Kitap Adı" F# ile fonksiyonel programlama, 'Yazar' Ali
Özgür """
// Çıktı
// "Kitap Adı" F# ile fonksiyonel programlama, 'Yazar' Ali Özgür
```

Çok uzun metinler tek satıra yazmak yerine birden fazla satır kullanarak da tanımlanabilir. Bu kullanım yönteminde metin normal olarak çift tırnak ikilisi veya çift tırnak üçlüleri arasına yazılır ve her satırın sonuna geri bölü \ karekteri konularak bir sonraki satırdan metne devam edilir. Çıktı oluşturulurken \ konulan satırdan sonra boşluk karakterleri göz ardı edilerek birden fazla satıra yayılmış olan metin tek satırda birleştirilir.

```

// Çok satıra yayılmış metin
let çokSatırlıMetin = " 1, \
                      2, \
                      3, "
// Çıktı
// 1,2,3
```

Diğer bir alternatif metin tanımlama yöntemi **verbatim** (motomot) metinlerdir. Verbatim metin tanımlamak için metnin başlangıcını ifade eden çift tırnak çiftinin önüne **@** karakteri konulur. Verbatim kelimesinin Türkçe karşılığından da anlaşılacağı üzere verbatim metinler ile kontrol karakterlerini** ** ile ifade etmeden birebir kullanabilirsiniz.

```
// Verbatim metin
let metin5 = @"Yazar \ Ali Özgür. Kontrol karakterleri \r \n \t \\"
// Çıktı
// Yazar \ Ali Özgür. Kontrol karakterlerimiz şunlar \r \n \t \\
```

3.3 Fonksiyonlar

Fonksiyonlar F#'in temelini oluşturan yapılarıdır. Fonksiyonların bir adı, girdi parametreleri, gövdesi ve çıktısı vardır. F#, fonksiyonların değer olarak kullanılabilmesi, isimsiz fonksiyonlar, fonksiyon girdi değerlerinin kısmi uygulanması ve fonksiyon kompozisyonu gibi fonksiyonel programlamanın özünü oluşturan işlemleri destekler.

F#'da fonksiyon tanımı basit değer ifadelerinde olduğu gibi "let" anahtar sözcüğü kullanılarak aşağıdaki formata uygun olarak yapılır

```
let <fonksiyon adı> <girdi1> <girdi2> ... <girdi N> =
    <fonksiyon gövdesi/kodu>

// Örnek fonksiyon tanımı
let topla x y =
    x + y
```

Örneğimizde

- topla : fonksiyonun adı
- x ve y : fonksiyonun girdi parametreleri
- x + y : fonksiyonun kodu yani gövdesi
- x + y ifadesinin sonucu : fonksiyonun çıktısı

Fonksiyon tanımı yapılırken girdi parametrelerinin ve çıktının tipinin tanımlanmasına genel olarak gerek duyulmaz, çünkü F# **tip çıkarsama** mekanizması sayesinde bu tipleri otomatik olarak çıkarsayabilir. Ancak, tipler kullanılmak istenirse fonksiyon tanımı aşağıdaki formata uygun olmalıdır.

```
let <fonksiyon adı> (<girdi1:tip>) ... (<girdi N:tip>) : <çıktı tipi> =
    <fonksiyon gövdesi/kodu>

// Örnek fonksiyon tanımı
let topla (x:int) (y:int) : string =
    sprintf "x + y = %d" (x+y)
```

Fonksiyon tanımı yapılırken tiplerin kullanımı opsiyoneldir. Örneğin girdi parametrelerinden sadece birkaçının tipi tanımlanabilir veya girdi parametre tipleri tanımlanmadan sadece çıktının tipi tanımlanabilir.

```
let topla (x:int) y : string =
    sprintf "x + y = %d" (x+y)

let topla' x y : string =
    sprintf "x + y = %d" (x+y)
```

F#'da bir fonksiyonun çıktısını döndürmek için diğer bazı dillerde olduğu gibi **return** benzeri bir anahtar kelime kullanımına ihtiyaç duyulmaz. Fonksiyonların çıktısı her zaman fonksiyon gövdesindeki son ifadenin değeridir.

```
let toplaVeÜçEkle x y =
    let yerel_değer = 3
    x + y + yerel_değer // Fonksiyon çıktısı, fonksiyon gövdesindeki son
ifade
```

Fonksiyonların çıktısı her zaman gövdesindeki son ifade ise çıktısı olmayan ve sadece yan etkisi için tasarladığımız fonksiyonların çıktısı ve çıktı tipi ne olur? Bu tür durumlarda **unit** adı verilen özel bir tip kullanılır. Bu tip C,C++,C# ve Java gibi dillerdeki **void** tipine çok benzer.

unit tipinden bir değer ifade etmek için boş çift parantez (**()**) kullanılır.

```
let toplaVeSadeceBas x y =
    let toplam = x + y
    printfn " İşlem sonucu x + y = %d" toplam
    ()
```

Yukarıdaki fonksiyon gövdesinde son ifade **()** olduğu için fonksiyonun çıktısı unit tipinden olacaktır. Aslında **()** ifadesi kaldırıldığında **printfn** ifadesi de unit tipinden bir değer döndürdüğü için dolaylı olarak toplaVeSadeceBas fonksiyonunun dönüş tipi de unit olur.

Fonksiyon gövdesindeki son ifadenin dönüş değerini kullanmadan fonksiyonun **unit** dönmesini sağlamak için F# standard kütüphanesi ile gelen **ignore** fonksiyonunu kullanılır.

```
// ignore normal fonksiyon olarak kullanımı
let topla x y =
    ignore (x+y)

// ignore |> operatörü ile kullanımı
let topla' x y =
    x + y |> ignore

// Alternatif yazım
let topla'' x y =
    let m = x + y
    ()
```

```
// Hatalı yazım
let topla''' x y =
    x + y
    ()
```

Örneğimizde **x + y** ifadesi hesaplanmasına ve int tipinde çıktı vermesine rağmen sonuç |> operatörü ile **ignore** fonksiyonuna aktarılır. Bu durumda fonksiyon gövdenizdeki son ifade **ignore** fonksiyonu çağırısıdır ve dönüş değeri **unit** tipindendir.

Fonksiyonların İmzası

Bir fonksiyonun imzası fonksiyonun girdi parametrelerinin ve çıktısının tiplerini tanımlamak için kullanılır. F#'da -> simgesi fonksiyonları matematiksel açıdan ele aldığımız bölümde tanımını yaptığımız **Tanım Kümesi**'nden **Değer Kümesi**'ne olan dönüşümü simgelemek için kullanılır. F# derleyicisinin veya etkileşimli yorumlayıcısını (FSI) çıktılarında fonksiyon imzaları aşağıdaki formata uygun olarak gösterilir.

```
      val fonksiyonAdı : tanım_kümesi -> değer_kümesi
```

```
// Tek parametreli fonksiyon
let kare x = sprintf "Karesi %f" (x**2.0)

// Çok parametreli fonksiyon
let topla x y = sprintf "Karesi %f" (x + y)
```

Yukarıdaki kod örneğinde ilk fonksiyon tanımını seçip Alt+ENTER kombinasyonu ile FSI'ya gönderdiğinizde

`val kare : x:float -> string` şeklinde bir çıktı alacaksınız.

Bu çıktı şu şekilde okunur; **kare** fonksiyonu **x** isimli **float** tipinden bir girdi parametresi alıp **string** tipinden bir çıktı üretir.

İkinci fonksiyon tanımı için ise

`val topla : x:float -> y:float -> string` şeklinde bir çıktı üretilir. Dikkat ederseniz girdi parametre sayısının artması imzada önemli bir değişikliğe neden olmadı, ifadenin soluna sadece ilave bir parametre tanımı eklendi.

Bilgi Kutusu (KURAL): Bu iki örneği genelleştirecek olursak; fonksiyon imzalarının en sağındaki tip fonksyionun çıktısının tipini gösterir, ifadenin solundaki diğer tipler ise girdi parametrelerini gösterir.

Şimdi gelin biraz daha karmaşık bir fonksiyon imzası örneği olarak **List.map** ifadesini FSI'da çalıştırdıktan sonra ürettiği çıktıyı inceleyelim. Çıktı olarak

```
      val it : (('a -> 'b) -> 'a list -> 'b list)
```

şeklinde bir fonksiyon imzası ile karşılaşırsınız. Bu imzayı sağdan sola şöyle okuruz; List.map öyle bir fonksiyondur ki

- En sondaki **'b list** ifadesine istinaden; çıktı olarak 'b tipinden elemanlar içeren bir liste döndürür
- **('a -> 'b)** ifadesine istinaden; ilk girdi paremetresi olarak 'a tipinden girdi alıp 'b tipinden çıktı üreten bir fonksiyon tipinde değer
- **'a list** ifadesine istinaden; ikinci girdi parametresi olarak ise 'a tipinden değerler içeren bir liste alır

Fonksiyon imzalarında fonksiyon tipinden parametreler çift parantez ile gruplanarak gösterilir.

Bilgi Kutusu (İPUCU): Bir fonksiyonun girdi parametre sayısı imza ifadesindeki **->** simgesi sayısı kadardır. **->** simgeleri sayılırken () ile gruplanmış fonksiyon tipi ifadelerindeki **->** simgeleri sayılmaz.

Değer Tipi Olarak Fonksiyonlar

F#'da ve diğer tüm fonksiyonel programlama dillerinde fonksiyonler birinci sınıf vatandaştırlar ve diğer basit ve karmaşık tipler gibi değer ifadelerinde tip olarak kullanılıp fonksiyonların girdisi veya çıktısı olarak tanımlanabilirler.

Fonksiyon tipli bir değer ifadesi tanımlamak için bir önceki başlıkta ayrıntılı bir şekilde ele aldığımız fonksiyon imzalarının formatına çok benzeyen aşağıdaki format kullanılır.

```
      let <değer_adı> : <tanım_kümesi> -> <değer_kümesi> = <>
```

Aşağıdaki örneğimizde **birArttır** isimli bir fonksiyon tanımlıyoruz. Bu fonksiyonun ilk parametresi string tipinden girdi alan ve hiçbirşey (unit) döndüren bir fonksiyon (string **->** unit tanımına istinaden) ikinci parametresi de x isimli int tipinden bir değer. Fonksiyonun gövdesinde toplama ifadesinden önce ve toplama yapıldıktan sonra **loglayıcı** fonksiyonu çağırılarak loglama yapılır.

```
(* 03_3_02.fsx *)
let birArttır (loglayıcı: string->unit) x =
    loglayıcı "İşleme başladım"
    let s = x + 1
    loglayıcı "İşlem tamam"
    s
let ekranaLogla (x:string)  =
    printfn "Log : %s" x

let dosyayaLogla (x:string) =
    // Dosyaya loglama kodu
    ()
birArttır ekranaLogla 42
birArttır dosyayaLogla 42
```

birArttır fonksiyonun **loglayıcı** fonksiyonunu parametre olarak almasındaki tasarımsal amaç fonksiyon kodunu değiştirmeden farklı loglama mekanizmalarının parametre olarak geşilebilen fonksiyonlar ile desteklenebilmesidir. Bu amaca uygun olarak **ekranaLogla** ve **dosyayaLogla** isimli iki fonksiyon tanımlanıyor. Bu fonksiyonların imzası (aslında tipi de denilebilir) **string -> unit** şeklinde olup **birArttır** fonksiyonun ilk parametresi olarak kullanılmaya uygundur.

Fonksiyon tiplerinin nasıl tanımlandığını ve kullanıldığını öğrendiğimize göre standard kütüphanedeki List modülü içinde bulunan **map** fonksiyonunu kendimiz oluşturmayı deneyelim. List.map fonksiyonun imzası şöyledir;

```
      val it : (('a -> 'b) -> 'a list -> 'b list)
```

Bu imzaya göre List.map fonksiyonu sonuç olarak da yeni bir liste döndürür ve ilk parametre olarak da bir fonksiyon alır. Bu imzada henüz değinmediğimiz tek konu **'a** ve **'b** şeklindeki ifadeler. Şimdilik bu ifadelerin **herhangi bir tip** veya **jenerik bir tip** anlamına geldiğini bilmeniz yeterlidir.

```
(* 03_3_03.fsx *)

// Prosedürel yaklaşım
let map  (f:'a->'b) (liste : 'a list) : 'b list =
    let sonuç = seq{for x in liste -> (f x)}
    sonuç |> List.ofSeq

[1..10] |> map (fun x -> x * x)

// Öz yinelemeli fonksiyon kullanımı ile daha fonksiyonel bir yaklaşım
let rec map' (f:'a->'b) (liste : 'a list) : 'b list =
    match liste with
    | [] -> []
    | baş::geriKalanlar -> (f baş) :: (map' f geriKalanlar)

[1..10] |> map' (fun x -> x * x)
```

Yukarıdaki örneğimizde **map** fonksiyonu

- **'a->'b** imzasına sahip ve **f** isimli bir fonksiyonu ilk parametre olarak alır.
- İkinci parametre **liste** isimli ve tipi **'a list** ('a herhangi bir tipte değer barındıran liste) olan bir değer
- Çıktısı ise **'b list** tipinden bir değerdir

Fonksiyonun gövdesinde **liste** içindeki tüm değerler için **f** fonksiyonu çalıştırılır ve **f** fonksiyonunun çıktısının tipinde ('b) değerler barındıran yeni bir liste döndürülür.

Bilgi Kutusu (İPUCU): Kitabımızın online Git deposundaki 03_3_03.fsx dosyası içinde map fonksiyonun öz yinelemeli bir fonksiyon olarak yazılmış halini inceleyebilirsiniz.

Parametresiz Fonksiyon Tanımları

F#'da girdi parametresi almayan fonksiyonları tanımlarken çok dikkatli olmalısınız. Programlama dillerinin çoğunda girdi parametresi almayan bir fonksiyon oluştururken basitçe parametrelerin tanımlanmaması yeterlidir. Ancak, F#'da parametresiz fonksiyonlar oluşturulurken **unit** tipinden en az bir girdi parametresi tanımlanmalıdır. Parametresiz fonksiyonlar **fonksiyon_adı()** formatına uygun olarak unit tipinin değeri olan boş çift parantez ile çağırılmalıdır.

Örneğimizde **kare** ve **ikininKaresiniAl** isimli iki fonkisyon tanımlandığını düşünelim.

```
let kare x = x * x
let ikininKaresiniAl = kare 2
```

Örnekteki iki satır Alt+Enter ile FSI'da seçip çalışçalıştırıldığında aşağıdaki gibi bir çıktı göreceksiniz

```
val kare : x:int -> int

val ikininKaresiniAl : int = 4
```

İlk ifade fonksiyon ifade formatına uygundur. Ancak, ikinci ifade bir fonksiyon ifadesi değildir. İkinci ifade bir değerin imzasıdır.

Değer imzaları formatı **val değer_ifadesi_adı : değer_tipi = değer** şeklindedir. Değer ifadelerinde fonksiyonel manada tanım ve değer kümeleri arasında bir dönüşüm yapılmadığı için -> sembolü bulunmaz. Gelin şimdi hatalı olan **ikininKaresiniAl** fonksiyonunu **unit** değerini kullanarak doğru bir şekilde tanımlayalım.

```
let kare x = x * x
let ikininKaresiniAl() = kare 2
ikininKaresiniAl() // Fonkisyon çağırısı
```

Bu ifadeleri FSI'da çalıştırdığımızda niyetimize uygun olarak aşağıdaki çıktıyı alırız

```
val kare : x:int -> int

val ikininKaresiniAl : unit -> int

val it : int = 4
```

ikininKaresiniAl fonksiyonunu unit değeri **()** kullanmadan FSI kullanarak çağırmayı deneyelim

```
ikininKaresiniAl
```

Yukarıdaki çağırı sonrasında FSI aşağıdaki çıktıyı üretir

```
val it : (unit -> int) =
```

Bu çıktı fonksiyon imzasına benziyor ama aynı zamanda değer ifadesi imzasını da andırıyor değil mi? Gerçekte bu ifade bir fonksiyon değerinin ifadesidir, çünkü F#'da fonksiyonlar da birer değer ifadesi olarak kullanılabilir yani dilin birinci sınıf vatandaşlarıdır. Bu ifadede

- **it** otomatik üretilen ve varsayılan bir değerin adını ifade eder
- **(unit -> int)** ifadesi değer tipinin girdi olarak unit alan çıktı olarak da int döndüren bir fonksiyon tipi olduğunu belirtir
- ifadesi ise **ikininKaresiniAl** fonksiyonun bellekteki adresini simgeleyen otomatik üretilmiş bir yer tutucu değerdir

Gördüğünüz gibi F#'da hiç bir girdi parametresi almayan fonksiyonları hem tanımlarken hem de kullanırken çok dikkatli olmalısınız. Aksi durumda derleyicinin veya FSI'in verdiği kriptik hata

mesajlarını çözümlemeye çalışarak zaman kaybedebilirsiniz. Daha da kötüsü derleyici veya FSI herhangi bir hata mesajı vermeyeceği için hatalı çalışan kod yazmış olabilirsiniz.

İsimsiz/Anonim Fonksiyonlar (Lambda İfadeleri)

Girdi parametresi olarak başka bir fonksiyonu alabilen yüksek dereceli fonksiyonları çağırırken basit hesaplamalar için isimsiz fonksiyon ifadelerini parametre olarak kullanabilirsiniz. Bu tür isimsiz fonksiyonlara **anonomi** fonksiyonlar denir.

Anonim fonksiyonlar aşağıdaki formata uygun oluşturulur

```
      fun <girdi1> <girdi2> ... <girdiN> -> <fonksiyon gövdesi>
```

Ananonim fonksiyonlarda girdi değerleri ve çıktı değerinin tiplerinin kullanılması ile ilgili kurallar isimli fonksiyonlar ile aynıdır.

```
(* 03_3_01.fsx *)

// 1.0 ile 10.0 arasındaki sayıların listesi
let list = [1.0..10.0]

// Kare fonksiyonu
let kare x = x**2.0

// Kare fonksiyonu kullanarak listedeki elemanların karesini alma
list |> List.map kare

// Anonim fonksiyon kullanarak listedeki elemanların karesini alma
list |> List.map (fun x -> x**2.0)
```

List modülündeki **map** fonksiyonu yüksek dereceli bir fonksiyondur çünkü ilk girdi parametresi olarak başka bir fonksiyon alır ve ikinci girdi parametresi olarak verilen listedeki tüm elemanları için ilk girdi parametresi olan fonksiyonu çalıştırır.

3.4 Fonksiyonların İleri Seviye Kullanımı

Fonksiyon İçinde Fonksiyon Tanımı

F#, fonksiyonlarınızın içinde yerel fonksiyonlar tanımlamanıza izin verir. Bu tür fonksiyon tanımlarına **iç içe fonksiyon** (nested function) denir. İç içe tanımlanan fonksiyonlar doğrudan kabuk fonksiyonun girdi parametrelerine ve kabuk fonksiyon içinde tanımlı yerel değer ifadelerine erişebilirler.

Bir fonksiyon içinde tekrar eden ve sadece o fonksiyona özgü kodu yerel bir fonksiyon tanımlayarak farklı yerlerde çağırabilirsiniz. F# dil seviyesinde bize sunduğu bu imkanı aslında arka planda derleyici seviyesinde kendisi de otomatik olarak kullanır. Şöyle ki; F#'da derleyici seviyesinde tüm fonksiyonlar tek parametreli fonksiyonlar olarak yeniden düzenlenir. Tanımladığımız birden fazla parametreli fonksiyonlar derleyici tarafından tek parametreli kabuk bir fonksiyon ve bunun içinde yer alan birden fazla yerel fonksiyon olarak yeniden organize edilip o şekilde derlenir. Bu reorganizasyon yeteneğini mümkün kılan ise iç fonksiyonların kabuk fonksiyonun tüm parametre ve yerel değerlerine erişebiliyor olmasıdır.

```
(* 03_3_04.fsx *)

// küp fonksiyonu ana fonksiyonumuz
let küp x =
    // küp içinde kare isimli yerel bir fonksiyon tanımlıyoruz
    let kare() =
        printfn "Yerel fonksiyon : Kare hesaplanıyor"
        x * x
    printfn "Ana fonksiyon : Küp hesaplanıyor"
    // yerel küp fonksiyonunu ana fonksiyon içinden çağırıyoruz
    kare() * x

küp 2

//Hatalı kullanım, kare fonksiyonu küp içindeki yerel bir fonksiyon
//kare()
```

Yukarıdaki örneğimizde **küp** fonksyionu içinde **kare** isimli parametresiz bir fonksiyon tanımlayıp **küp** içinden bunu kullanıyoruz. Yerel bir fonksiyon olan **kare** fonksiyonunu ana fonksiyon olan **küp** dışındaki bir kod alanında kullanamayız.

Currying

Currying teriminin tam olarak Türkçe bir karşılığı yok, çünkü bu terim fonksiyonel programlama dillerinin ortaya çıkmasına ve gelişmesine önemli katkıları olan ünlü matematikçi **Haskell Curry**'nin anısına ortaya atılmış bir terimdir. Currying denilen yöntem istisnasız tüm fonksiyonel programlama dilleri tarafından hem dil hem de derleyici/yorumlayıcı seviyesinde uygulanan bir yöntemdir.

Bu yöntem aşağıdaki prensipler sayesinde mümkündür.

1. Çok parametreli fonksiyonlar tek parametreli ana bir fonksiyon ve iç içe geçmiş tek parametreli fonksiyonlar olarak düzenlenebilir
2. Fonksiyonlar başka fonksiyonlara girdi parametresi olarak geçilip çıktı olarak döndürülebilir
3. Yerel fonksiyonlar ana fonksiyonun parametrelerine erişebilir
4. Ana fonksiyonun çıktısı tek parametre alan yerel bir fonksiyon olabilir

Aşağıdaki örneğimizde iki parametreli bir fonksiyonun Currying yöntemi ile nasıl ifade edildiğini görebilirsiniz.

```
(* 03_3_05.fsx *)

// İki parametreli fonksiyon tanımı
let ikiDeğeriEkranadaGöster x y =
    printfn "Değerler x=%d, y=%d" x y

// Test
ikiDeğeriEkranadaGöster 1 2

// Tek parametreli fonksiyon olarak tanımlama
let tekDeğeriEkrandaGöster x =
    // Yerel fonksiyon
    let _ikiDeğeriEkrandaGöster y =
         printfn "Değerler x=%d, y=%d" x y

    // Yerel fonksiyonu ana fonksiyonun çıktısı olarak dön
    _ikiDeğeriEkrandaGöster

// Test

// Aşağıdaki ifadenin sonucu (int->unit) imzalı bir fonksiyon
// 1 parametresi tekDeğeriEkrandaGöster çıktısı olan fonksiyona gömülür
let  ikiDeğeriEkrandaGöster' = tekDeğeriEkrandaGöster 1

// ikiDeğeriEkrandaGöster' tek parametre alan bir fonksiyon
// 2 parametresi ile çağırırsak sonuç ikiDeğeriEkranadaGöster ile aynı
olur
ikiDeğeriEkrandaGöster' 2
```

Yukarıdaki kod parçasında önce **ikiDeğeriEkranadaGöster** isimli iki parametreli normal bir fonksiyon tanımlanır. Daha sonra da normal fonksiyonun yaptığı işlemin tek parametreli fonksiyonlar ile nasıl yapılacağını gösteren **tekDeğeriEkrandaGöster** ana fonksiyonu ve **ikiDeğeriEkrandaGöster'** fonksiyon değeri tanımlanıyor.

- **tekDeğeriEkrandaGöster** tek parametreli bir fonksiyondur
- tekDeğeriEkrandaGöster içinde **_ikiDeğeriEkrandaGöster** isimli parametreli yerel bir fonksiyon tanımlanır

- **_ikiDeğeriEkrandaGöster** yerel fonksiyonu ana fonksiyonun **x** parametresine doğrudan erişebilir. Bu nedenle ekrana basma işlemini bu fonksiyona yaptırıyoruz.
- Ana fonksiyonun dönüş değeri **_ikiDeğeriEkrandaGöster** fonksiyonu olarak tanımlanır
- **ikiDeğeriEkrandaGöster'** isimli bir değer ifadesi tanımlanır. Bu değer ifadesi **tekDeğeriEkrandaGöster 1** çağırısının sonucu olan fonksiyon değerini tutar. Dikkat ederseniz bu çağırı **tekDeğeriEkrandaGöster** yerel fonksiyonunu çalıştırmaz onu çıktı olarak döndürür
- **ikiDeğeriEkrandaGöster'** fonksiyonu **2** parametresi ile tekrar çağırılır. Bu durumda yerel **tekDeğeriEkrandaGöster** fonksiyonu 2 parametresi ile çalıştırılır ve ekrana görmek istediğimiz ifade çıkar

Aşağıdaki örnek ile 3 parametreli bir fonksiyonun tek parametreli fonksiyonlar şeklide nasıl ifade edildiğini görebilirsiniz.

```
(* 03_3_06.fsx *)

// Ana fonksiyon
let üçSayıyıÇarp x =
    // Ana fonksiyon içinde yerel fonksiyon
    let çarp' y =
        // Yerel fonksiyon içinde yerel fonksiyon
        let çarp'' z = x * y * z
        // Yerel fonksiyon kendi içindeki yerel fonksiyonu döndürür
        çarp''
    // Ana fonksiyon kendi içindeki yerel fonksiyonu döndürür
    çarp'

// İlk çağırı, val çarp' : (int -> int -> int) imzalı
// fonksiyon döndürür. çarp' 3 değerini içinde barındırır
let çarp' = üçSayıyıÇarp 3

// İkinci çağırı. val çarp'' : (int -> int) imzalı
// fonksiyon döndürür. çarp'' hem 3 hem de 4 değerini içinde barındırır
let çarp'' = çarp' 4

// Son çağırı. çarp'' 3 ve 4 değerini içinde barındırıyor
// 5 değeri de verilince sonuç olarak 120 hesaplanıyor
çarp'' 5

// Üç parametreli normal bir fonksiyon tanımı
let üçSayıyıÇarp' x y z = x * y * z
```

Curried fonksiyonların imzası

Üç sayının çarpımı örneğinde **üçSayıyıÇarp** fonksiyonun imzası FSI tarafından şu şekide ifade edilir

```
val üçSayıyıÇarp : x:int -> (int -> int -> int)
```

Bu imza **üçSayıyıÇarp** fonksiyonunun **int** tipinden tek girdi parametresi olan ve çıktı olarak **(int -> int -> int)** imzasına istinaden

- İki tane **int** parametre alan ve
- Çıktı olarak **int** döndüren bir fonksiyon

döndürdüğünü ifade eder.

Örneğimizdeki **üçSayıyıÇarp'** isimli 3 parametreli normal fonksiyon tanımını FSI'da çalıştırdığımızda ise şöyle bir fonksiyon imzası görürüz

```
val üçSayıyıÇarp' : x:int -> y:int -> z:int -> int
```

Bu imzanın **üçSayıyıÇarp** imazasından tek farkı () ile gruplanmış 3 parametrelik bir ifadenin varlığıdır. Pratikte parantezlerin olması ile olmaması arasında önemli bir fark yoktur. Ancak, kod yazarken hatalı kullanıma mahal vermemek için () ile gruplanmış ifadelerin girdinizin veya çıktınızın basit tipli bir değer değil fonksiyon değeri olduğunu belirttiğini unutmayın.

Bilgi Kutusu (DİKKAT!): Çok parametreli fonksiyonları eksik parametre ile çağırmanız durumunda F# derleyicisi derleme anında hata vermez. Ancak programınız çalışma anında eksik parametreli çağrılarınız nedeni ile hata durumuna düşebilir. Bu nedenle çok parametreli fonksiyon çağrıları yaparken dikkatli olmalısınız. Çok parametreli fonksiyonlar eksik parametre ile çağırıldığında sonuç olarak basit bir değer değil bir fonksiyon değeri döndürülür.

```
(* 03_3_07.fsx *)
(* Eksik parametreli fonksiyon çağırlarına DİKKAT *)

// Fonksiyon tanımı
let sayılarıEkrandaGöster x y z = printfn " x = %d, y=%d, z=%d" x y z

// Fonksiyonu eksik iki parametre ile çağırdık.
// 2 sayısı ekranda gösterilmez bunun yerine FSI aşağıdaki gibi bir çıktı
üretir
// val it : (int -> int -> unit)
// Bu çıktıya göre "sayılarıEkrandaGöster 2" çağırısı iki int girdi
// parametresi alıp int tipinden bir sonuç döndüren bir fonksiyon döndürür
// Ekrana hiç birşey basılmaz
sayılarıEkrandaGöster 2
```

Fonksiyonlarda Kısmi Uygulama Yöntemi

F#, fonksiyonel programlama dillerinin hepsinde olduğu gibi, bir fonksiyonun bazı parametrelerini sabitleyip yeni bir fonksiyon oluşturmak için **kısmi uygulama** (partial application) desteği sunar. Kısmi uygulama önceki başlıkta ele aldığımız Currying sayesinde mümkündür.

```
(* 03_3_08.fsx *)

let ekle x y = x + y

// İmzası val birEkle : (int -> int) olur
// int çıktı veren ve tek int girdi alan bir fonksiyon
let birEkle = ekle 1
birEkle 42

let çarp x y = x * y
// İmzası val ikiİleÇarp : (int -> int) olur
// int çıktı veren ve tek int girdi alan bir fonksiyon
let ikiİleÇarp = çarp 2
ikiİleÇarp 42
```

Örneğimizde, **birEkle** değeri **ekle** fonksiyonunun tek bir parametre ile çağırılması sonucunda oluşturulan bir fonksiyondur. **ekle 1** şeklindeki ifade ile iki parametreli **ekle** fonksiyonun birinci parametresini **1** değeri ile sabitleyip ikinci parametreyi boşta bırakıyoruz. Bu durumda **birEkle** fonksiyonunun imzası **(int -> int)** olur. **birEkle** int bir parametre alıp int çıktı üreten bir fonksiyondur.

Kısmi uygulama ile basit tipli fonksiyon parametrelerini sabitleyebildiğimiz gibi fonksiyon tipinde parametreleri de sabitleyebiliriz.

```
(* 03_3_09.fsx *)

// Basit bir "kare" fonksiyonu
let kare x = x * x


// List.map fonksiyonun ilk parametresi "kare" fonksiyonu
// ikinci parametresi olan liste verilmemiş

// kareleriniAl imzası şöyle olur
// "val kareleriniAl : (int list -> int list)"
// kareleriniAl girdi olarak int değer listesi alıp çıktı olarak
// int değer listesi döndüren bir fonksiyondur
let kareleriniAl = List.map kare

// "kareleriniAl" [1..10] listesi parametresi ile çağırılır
kareleriniAl [1..10]
```

Bu örneğimizde önce **kare** isimli basit bir fonksiyon tanımladık. Daha sonra da **List.map** fonksiyonun ilk parametresi olarak bu **kare** fonksiyonunu geçerek sabitledik, List.map'in beklediği ikinci

parametreyi vermedik. Sonuçta **kareleriniAl** isimli girdi olarak int listesi alıp çıktı olarak da int listesi döndüren bir fonksiyon değeri oluşturduk. **kareleriniAl** fonksiyonunu [1..10] listesini parametre olarak kullanarak çalıştırdık.

Kısmi Uygulama Uyumlu Fonksiyon Tasarımı

Kısmi uygulama yönteminin alameti farikası bazı girdi parametrelerinin değerlerinin sabitlenmesi geri kalanının ise hiç belirtilmemesidir. Kısmi uygulama ile var olan fonksiyonlardan daha özelleşmiş yeni fonksiyonlar oluşturulabilir.

Esnek kullanımı hedefleyen fonksiyonlar kodlarken aşağıdaki iki soruya odaklanıp kodumuzu bu soruları cevaplayacak şekilde yazmalıyız.

1. Fonksiyonun kendi başına yaptığı çok önemli iş nedir
2. Fonksiyon, bu çok önemli işi yapması için dışarıdan hangi veri ve fonksiyonlara ihtiyaç duyar

Örneğin F# standard kütüphanesindeki **List** modülünü bir fonksiyonu olan **map** fonksiyonu için bu iki sorunun cevabını şöyle verebiliriz

1. Fonksiyonun yaptığı önemli iş bir liste üzerinde her bir elemanı sırasıyla ziyaret ederek bir fonksiyon çalıştırmaktır
2. Bu önemli işi yaparken de üzerinde döneceği bir liste ve bu liste üzerinde dönerken çalıştıracağı bir fonksiyona ihtiyaç duyar

Fonksiyonun girdi parametrelerinin ne olduğu kadar sıralaması da önemlidir, çünkü kısmi uygulama yönteminde sabitlenen parametreler fonksiyonun soldan sağa ilk parametreleridir geveşek bırakılan parametreler ise fonksiyonun son parametresidir. Bu kısıtlama çerçevesinde hangi parametrelerin başta hangisinin sonda geleceğini neye göre ve nasıl belirlenir?

- Static parametreler, yani değeri değişmeyen parametreler başta
- Özelleşmiş veri tipleri veya liste gibi birden çok eleman barındıran dinamik tipli parametreler de bunlardan sonar gelecek şekilde

fonksiyon tanımı yapılmalıdır. Bu iki kuralı aklınızda tutamazsanız kısa yol olarak şöyle basit bir yaklaşım uygulayabilrisiniz; |> (ileri aktarım) operatörünün solunda olması istenen parametreler her zaman fonksiyon tanımında son parametre olarak yer almalıdır.

Aşağıdaki örnekte **List.map** fonksiyonunun tanımını ve bu fonksiyon ile nasıl kısmi uygulama yapıldığını inceleyebilirsiniz.

```
(* 03_3_10.fsx *)

(*
    List.map fonksiyonun imzası şöyledir;
    val it : (('a -> 'b) -> 'a list -> 'b list)
    Bu fonksiyon ilk parametre olarak bir fonksiyon ikinci parametre
olarak
    ise bir liste alır
*)

// "küp" fonksiyonumuz
let küp x = x * x * x

// test listemiz 1 ile 10 arasındaki değerleri barındırır
let liste = [1..10]

// List.map fonksiyonun normal kullanımı
List.map küp liste

// List.map fonksiyonun ileri akatrım operatörü ile kullanımı
liste |> List.map küp

// List.map fonksiyonunndan faydalanan hepsininKüpünüAl isimli
// yeni bir fonksiyonu List.map'i kısmi uygulayarak oluşturuyoruz
// Kısmi uygulamada List.map fonksiyonu için ilk parametreyi "küp"
// fonksiyonu olacak şekilde sabitledik ikinci parametre belirtilmedi

let hepsininKüpünüAl = List.map küp

//Türettiğimiz hepsininKüpünüAl fonksiyonun imzası şöyledir
// "val hepsininKüpünüAl : (int list -> int list)"

// Bu yeni fonksiyon bir int listesi alır ve
// sonuç olarak bir int listesi döner
hepsininKüpünüAl liste

// İleri aktarım operatörü ile çağırıyoruz
liste |> hepsininKüpünüAl
```

Şimdi gelin kendi geliştireceğimiz iki adet **map** fonksiyonu ile kısmi uygulama uyumluluğunun neden önemli olduğunu inceleyelim.

```
(* 03_3_11.fsx *)

let küp x = x * x * x
let liste = [1..10]

//------ KISMİ UYGULAMA UYUMLU OLMAYAN YAKLAŞIM ------//
// Paremetreleri kısmi uygulama için uygun sıralanmamış
// map' fonksiyonu tanımı
// İlk parametre bir liste
// Son parametre bir fonksiyon
let map' liste f =
    let sonuç = seq{for x in liste -> (f x)}
    sonuç |> List.ofSeq


// map' fonksiyonunu kullanarak bir listenin tüm değerlerinin
// küpünü alacak yeni bir fonksiyonu aşağıdaki gibi türetemiyoruz
// let hepsininKüpünüAl = map' küp

// map' fonksiyonun kullanarak ancak aşağıdaki gibi bir
// hepsininKüpünüAl fonksiyonu oluşturabiliriz
let hepsininKüpünüAl liste = map' liste küp

hepsininKüpünüAl liste
liste |> hepsininKüpünüAl

//------ KISMİ UYGULAMA UYUMLU YAKLAŞIM ------//

// Paremetreleri kısmi uygulama için uygun sıralanmış
// map'' fonksiyonu tanımı.
// İlk parametre bir fonksiyon
// Son parametre bir liste
let map'' f liste =
    let sonuç = seq{for x in liste -> (f x)}
    sonuç |> List.ofSeq
```

```
// map'' fonksiyonunu kullanarak bir listenin tüm değerlerinin
// küpünü alacak şekilde yeni bir fonksiyonu aşağıdaki gibi
// türetebiliriz
let hepsininKüpünüAl' = map'' küp

hepsininKüpünüAl' liste
liste |> hepsininKüpünüAl'
```

Öz Yinelemeli Fonksiyonlar

Kendi kendini çağıran fonksiyonlara **öz yinelemeli** fonksiyonlar denir. F#'da öz yinelemeli bir fonksiyon tanımlamak için **rec** anahtar kelimesi kullanılır. Öz yinelemeli fonksiyon tanımı **rec** kullanımı dışında normal fonksiyon tanımlama şablonu ile aynı şekildedir.

```
      let rec fonksiyon_adı girdi1 ... girdiN = fonksiyon_kodu
```

Öz yinelemeli fonksiyonlar için **fibonacci sayıları** ve **faktöriyel** hesaplaması klasik örnekler olarak literatürde kendilerine yer edinmiştir. Şimdi gelin bu iki kavramı tanımlayıp fonksiyonlarını F# ile oluşturalım.

Fibonacci Sayıları: Her bir Fibonacci sayısının kendinden önceki iki Fibonacci sayısının toplamı olduğu pozitif tam sayı dizisidir. Formel olarak; n. Fibonacci sayısı $\mathbf{F_n = F_{n-1} + F_{n-2}}$ şeklinde ifade edilir. Örneğin; 1, 1, 2, 3, 5, 8 şeklinde devam eden dizi Fibonacci Sayıları dizisidir ve 4. Fibonacci sayısının değeri olan 3 kendinden önceki 2 ve 1'in toplamına eşittir.

```
(* 03_3_12.fsx *)

// Fibonacci Sayısı
// F□ = F□₋₁ + F□₋₂
let rec fibonacci n =
    if n <= 1 then
        n
    else
        fibonacci (n-1) + fibonacci(n-2)

// TEST : 4. fibonacci sayısının değeri
fibonacci 4

// TEST : 1 ile 10 arasındaki Fibonacci sayıları
[1..10] |> List.iter ( fun x -> printfn "%d. fibonaci sayısı = %d" x (
fibonacci x))
```

Faktöriyel Hesaplama: Bir sayının faktöriyeli 1 ile kendisi arasındaki pozitif tam sayıların çarpımının sonucudur ve **n!** olarak ifade edilir. Örneğin; 5! = 5*4*3*2*1 = 120 olarak hesaplanır.

```
(* 03_3_12.fsx *)

// Faktöriyel Hesaplama
// n! = n * (n-1) * (n-2) * .... * 1
let rec faktöriyel n =
    if n < 1 then
        1
    else
        n * faktöriyel(n-1)

// TEST : 6'nın faktöriyeli
faktöriyel 6

// TEST : 1 ile 10 arasındaki sayıların faktöriyeli
[1..10] |> List.iter ( fun x -> printfn "%d! = %d" x ( faktöriyel x))

(*
// Sonlanma koşulu olmayan hatalı öz yinelemeli fonksiyon
let rec fibonacci' n =
    fibonacci (n-1) + fibonacci(n-2)

fibonacci' 2147483647 //En büyük işaretli 32-bit tam sayı
*)
```

İç içe fonksiyon çağırılarında program akış kontrolü, fonksiyon girdi parametreleri ve fonksiyon dönüş değerleri işletim sistemi tarafından **yığın** (stack) adı verilen veri yapısı kullanılarak takip edilir. Örneğin A fonksiyonu B fonksiyonunu çağırsın. A fonksiyonu çalışmaya başlayıp B'nin çağırıldığı satıra gelindiğinde işletim sistemi yığına

1. B'nin çalışması bittiğinde A'nın nereden devam edeceğini hatırlatmak için bir **işaretçi değeri** ve
2. A'nın B'ye geçtiği **girdi parametrelerinin** değerlerini koyar

B çalışıp sonlandığında ise yığına sonuç değeri konulur. B'nin çalışmasının bitmesi ile birlikte işletim sistemi A fonksiyonunu yığındaki işaretçinin gösterdiği yerden itibaren çalıştırmaya devam eder.

```
let B x =
    printfn "B fonksiyonu, değer = %d" x

let A() =
    printfn "A fonksiyonu başladı"
    B 12 // B fonksiyonu çağırılıyor
    printfn "A fonksiyonu tamamlandı"

// A çağırılıyor
A()
```

Öz yinelemeli fonksiyonlar tanımları gereği kendilerini çağırırlar ve her çağrı ile birlikte bir önceki çağrının nereden devam edeceğini gösteren işaretçi değeri yığın'a eklenir. Yığın veri yapısının boyutu genel anlamda tüm işletim sistemlerinde kısıtlıdır. Örneğin, 64-bit Windows için yığın boyutu 4 MB iken çoğu Linux dağıtımı için bu değer 8 MB olarak tanımlıdır. Yığın boyutunun sabit ve kısıtlı olması yığına konulabilecek değerlerin, başka bir ifadeyle iç içe çağırılabilecek fonksiyon sayısının, sonlu olduğu anlamına gelir. Yığın ile ilgili boyut kısıtlaması normal fonksiyonların birbirini çağırması gibi durumlarda soruna neden olmaz, çünkü yığında kısıtlı boyutuna rağmen pratikte ulaşılması zor olan miktarda fonksiyon çağırısı takip edilebilir.

Öz yinelemeli fonksiyonlarda fonksiyonun bitiş koşulunun varlığı çok önemlidir, çünkü bu fonksiyonlar eğer bir bitiş koşulu olmazsa sonsuz döngü şeklinde kendi kendini çağırırı ve yığın'ın tüm kapasitesini tüketebilir. Bu nedenle, öz yinelemeli fonksiyonlarda girdi parametreleri tarafından belirlenen ve fonksiyonun kendini çağırmadığı bir sonlanma koşulu tanımlanmalıdır.

Fibonacci sayısını hesaplayan fonksiyon örneğimizde girdi parametresinin değeri 1 veya daha küçük bir sayı ise fonksiyon kendini çağırmaz ve bir değer döndürür. Fonksiyonun sonlanma koşulu yoksa veya mümkün olmayan bir koşul kodlandıysa fonksiyon çağırısı yığın kapasitesinin tükenmesi nedeni ile **yığın taşma** (stack overflow) hatası verir ve program sonlanır.

```
// Sonlanma koşulu olmayan hatalı öz yinelemeli fonksiyon
let rec fibonacci' n =
    fibonacci (n-1) + fibonacci(n-2)

fibonacci' 2147483647 //En büyük işaretli 32-bit tam sayı
```

Döngü Yapıları Olarak Öz Yinelemeli Fonksiyon Kullanımı

Prosedürel dillerde döngü oluşturmak için kullanılan **for** ve **while** yapılarını fonksiyonel programlama ilkelerine uygun olarak kolayca kurgulayabilirsiniz.

Önce C#'da (prosedürel .NET dillerinden birisi) basit bir **for** döngüsünü nasıl tanımlayabileceğimize bir göz atalım.

```
// C# for döngüsü
void Main()
{

    for(int i=5; i>=0;i--) // döngü ve i sayacı
    {
        // Döngü gövdesi başlangıcı

        Console.WriteLine("Döngü, sayaç = {0}", i);

        // Döngü gövdesi bitişi
    }
}
```

Şimdi de F# kullanarak C#'daki **for** döngü yapısını nasıl oluşturabileceğimizi inceleyelim.

```
(* 03_3_14.fsx *)

// Öz yinelemeli döngü fonksiyonu
let rec döngü f sayaç =

    if sayaç = 0 then
        () // Bitiş koşulu, sayaç sıfır ise unit döndür
    else
        //G irdi olarak verilen fonksiyonunu
        // sayaç değeri ile çağır
        f(sayaç) // f fonksiyonunu sayaç parametresi ile çağır
        döngü f (sayaç-1) // tekrar döngü çağır


// TEST
let sayaç = 5
döngü (fun i-> printfn "Döngü, sayaç = %d" i) sayaç
```

Örneğimizde **döngü** isimli bir öz yinelemeli fonksiyon tanımlıyoruz. Bu fonksiyonun ilk parametresi her bir dönüşte çalıştırılması istenen kodu temsil eden bir **f** fonksiyonu, ikinci parametre ise bitiş koşulunu kontrol eden **sayaç** değeridir. Fonksiyonel dillerde iafdelerin değerleri değişmez (immutable) olduğu için C#'da yaptığımız gibi bir **i** sayaç değişkenin değerini azaltmak yerine öz yinelemeli fonksiyon çağırısında **sayaç** değerinin bir eksiğini bir sonraki çağırıya parametre olarak geçiyoruz.

Aşağıda yine C# için **while** koşullu döngü yapısını nasıl kullanıldığını görebilirsiniz. **while** döngüsü koşul sağlandığı sürece döngü gövdesindeki kodun çalışmasını sağlamak için kullanılır.

```
// C# while döngüsü
void Main()
{
    Func<DateTime,bool> üçSaniyeBittiMi = (x) => {
        return System.DateTime.Now - x <= System.TimeSpan.FromSeconds(3);
    };

    var başlangıçZamanı  = System.DateTime.Now;
    while(üçSaniyeBittiMi(başlangıçZamanı)) // döngü ve koşul cümlesi
    {
        // Döngü gövdesi başlangıcı

        Console.WriteLine("Koşullu döngü, başlangıç = {0}, şu an =
{1}",başlangıçZamanı, System.DateTime.Now);
        // Döngü gövdesi bitişi
    }
}
```

Şimdi de F# kullanarak **while** koşullu döngü yapısını nasıl oluşturabileceğimizi inceleyelim.

```
// Koşullu döngü öz yinelemeli fonksiyonu
let rec koşulluDöngü koşul f =
    if koşul() then // Koşul doğru ise
        f() // Önce fonksiyonu çağır
        koşulluDöngü koşul f // Tekrar koşulluDöngü çağır
    else
        () // Koşul doğru değil unit dön ve sonlan

// TEST
let şuAn() = System.DateTime.Now
let üçSaniyeBittiMi x =
    şuAn() - x <= System.TimeSpan.FromSeconds(3.0)
let  başlangıçZamanı  = şuAn()
koşulluDöngü
    ( fun() -> üçSaniyeBittiMi başlangıçZamanı )
    ( fun() ->
        printfn "Koşullu döngü, başlangıç = %A, şu an = %A"
başlangıçZamanı System.DateTime.Now
    )
```

Örneğimizde **koşulluDöngü** isimli bir öz yinelemeli fonksiyon tanımlıyoruz. Bu fonksiyonun ilk parametresi her bir dönüşte kontrol edilmesi istenen koşulu hesaplayan bir **koşul** fonksiyonu, ikinci parametre de koşul doğru ise çalıştırılacak olan **f** fonksiyonudur.

Bilgi Kutusu (DİKKAT!): Öz yinelemeli fonksiyonlar algoritmaları pratik bir şekilde kodlamak için oldukça kullanışlı yapılardır. Ancak, bu yapıları kullanırken performans karakteristikleri iyi analiz edilmeli ve bitiş koşulunun çalışması garanti altına alınmalıdır. Ayrıca, öz yinelemeli fonksiyonların okunması ve anlaşılması normal fonksiyonlara göre biraz daha zahmetlidir.

Karşılıklı Öz Yinelemeli Fonksiyonlar

Bazı fonksiyonlar kontrollü bir şekilde ve karşılıklı olarak birbirini çağırabilir. Bu tür fonksiyonlara **karşılıklı öz yinelemeli** (mutually recursive) fonksiyonlar denir.

F#'da dosyaların ve dosyalar içindeki fonksiyon, tip ve modül tanımlarının sırası önemlidir. Örneğin, bir fonksiyonun başka bir fonksiyon tarafından çağırılabilmesi için çağırıldığı kod satırından önce tanımlanmış olması gerekir. Aşağıdaki örneğimizde **A** fonksiyonu içindeki **B()** çağırısı geçersizdir, çünkü **B** fonksiyonu o noktada henüz tanımlı değildir.

```
(* 03_3_13.fsx *)

// ------------------- HATALI TANIM -------------------
(*
    Aşağıdaki karşılıklı öz yinelemeli fonksiyon tanımı
    hatalıdır. Derleyici bu fonksiyonların tanımını
    derlemez hata verir.
*)

// A fonksiyonu
let A() =
    B()

// B fonksiyonu
let B() =
    A()
```

Karşılıklı öz yinelemeli fonksiyonlar şöyle oluşturulur; birinci fonksiyon tanımı

```
      let rec fonksiyon_adı parametreler =
```

yapısı ile başlar ve normal bir öz yinelemeli fonksiyon olarak tanımlanır. İlk fonksiyon dışındaki fonksiyonlar ise

```
      and fonksiyon_adı parametreler =
```

şablonuna uygun olarak tanımlanır.

```
(* 03_3_13.fsx *)
// ------------------- DOĞRU TANIM -------------------
let rec Çift x =
    if x = 0 then
        true
    else
        Tek (x-1)
and Tek x =
    if x = 0 then
        false
    else
        Çift (x-1)

// TEST
Çift 1
Çift 2

Tek 3
Tek 4
```

Örneğimizde karşılıklı öz yinelemeli **Çift** ve **Tek** fonksiyonları tanımlanmıştır. **Çift** fonksiyon tanımı

```
      let rec Çift x =
```

şeklinde başlar ardından da **Tek** fonksiyonu

```
      and Tek x =
```

şeklinde tanımlanır.

```
Tek 3
  Çift 2
     Tek 1
        Çift 0 -> true

Çift 4
    Tek 3
        Çift 2
            Tek 1
                Çift 0 -> true
```

Bilgi Kutusu (DİKKAT!): Öz yinelemeli fonksiyonlar tanımlanırken birinci fonksiyondan sonraki fonksiyon tanımları ilk fonksiyon tanımı ile aynı miktarda girintili olarak hizalanıp yapılır. F#'da girintiler ile hizalamanın kod alanlarını belirlediğini unutmayın.

3.5 Temel Veri Tipleri

Basit veri tipleri (tam sayı, ondalık sayı, metin, karakter vs) ile bir çok işlem yapılabilir. Ancak, sadece bu tipler kullanılarak daha karmaşık programların yazılması mümkün değildir. F#'da bu basit veri tiplerine ilave olarak aşağıdaki **temel tipler** de bulunur.

- **unit**
- **tuple** (Değer Grubu)
- **list** (Liste)
- **option** (opsiyon)

Unit

F# ile ister basit bir değer ifadesi tanımlayın isterseniz bir fonksiyon tanımlayın istisnasız tüm ifadelerin bir değeri olmalıdır. İlk bakışta bu çok kısıtlayıcı bir kural gibi görünür, çünkü değeri olmayan veya hiç bir değer döndürmeyen fonksiyonların tanımlanmasını imkansız kılar. Ancak, biliyoruz ki **printfn** gibi bazı standard kütüphane fonksiyonları herhangi bir değer döndürmez ve yalnızca yan etkileri için kullanılır.

F#'da **hiç bir şey** değerini ifade etmek için adı **unit**, değeri **()** (çift parantez) olan bir tip kullanılır. **unit** tipini C, C++, Java veya C# gibi prosedürel dillerdeki **void** tipine benzetebiliriz. Programlama dillerinde tipler kavramsal modelleme için kullanılır, bu nedenle **hiç birşeyi** modellemek için **unit** gibi bir tipin olması sizi şaşırtmamalı.

Bilgi Kutusu (AÇIKLAMA): printfn fonksiyonunun yan etkisi standard giriş/çıkış birimine (genelde ekrana) verilen metni yazmasıdır.

```
(* 03_5_01.fsx *)

// Değeri unit tipinden olan basit bir değer ifadesi
let değer = ()

// unit döndüren bir fonksiyon
let fonksiyon1 x =
    printfn "x'in değeri = %d" x
    ()

// Dolaylı olarak unit döndüren fonksiyon
// Son çağrı printfn'e ve printfn'in dönüş değeri unit
let fonksiyon1' x =
    printfn "x'in değeri = %d" x


// Parametresiz fonksiyon
// Aslında bu fonkisyon tipi unit olan
// tek parametreli bir fonksiyon
let fonksiyon2 () =
    printfn "Parametresiz fonksiyon"
    42
```

```fsharp
// Parametre değerlerini toplayan ancak
// sonucu yutan ve dönüş değeri olmayan bir fonksiyon
let fonksiyon3 x y =
    x + y |> ignore // toplama sonucu yutuldu
    printfn "Toplama yapıldı ancak sonuç yutuldu"


// Son parametresi unit tipinde olan fonksiyion
let fonksiyon4 x y z:unit =
    x + y |> ignore // toplama sonucu yutuldu
    printfn "Toplama yapıldı ancak sonuç yutuldu"

// TEST
fonksiyon1 42
fonksiyon1' 42

fonksiyon2()
fonksiyon3 42 0
fonksiyon4 42 0 ()
```

Null

F#'da doğrudan **null** değerler tanımlamak mümkün değildir. Ancak, eğer başka bir .NET dili ile geliştirilmiş bir kütüphane kullanıyorsanız ilgili kütüphaneden yapacağınız fonksiyon çağırıları **null** değer döndürebilir. Benzer şekilde diğer .NET dilleri ile geliştirlmiş kütüphane fonksiyonlarına parametre olarak **null** değerini gönderebilirsiniz.

```
(* 03_5_01_null*)

open System


// Kişi isimli kayıt tipi tanımı
type Kişi = {Ad:string;Soyad:string}

// Yeni bir kişi oluşturma
let kişi = {Kişi.Ad="Ali"; Kişi.Soyad = "Özgür"}

// kişi' ifadesine null değer vermek mümkün değil
//let kişi':Kişi = null

let tarihiÇözümle (str: string) =
    let (success, res) = DateTime.TryParse(str, null,
System.Globalization.DateTimeStyles.AssumeUniversal)
    if success then
        Some(res)
    else
        None

tarihiÇözümle "2017-09-25 10:00:00"
```

Örneğimizdeki **tarihiÇözümle** fonksiyonu standard .NET kütüphanesindeki **DateTime** sınıfı için tanımlı olan **TryParse** fonksiyonunu kullanıyor. Bu fonksiyonun ikinci parametresi olan **provider** değeri olarak null geçiliyor.

Eğer F# içinden oluşturduğunuz tiplerin değerlerinin **null** olmasına izin vermek istiyorsanız tipinizi **AllowNullLiteralAttribute** özelliği ile dekore etmeniz gerekir.

```
(* 03_5_01_null*)

// Araba isimli sınıf tanımı
[<AllowNullLiteral>]
type Araba (marka:string,model:string,modelYılı:int) =
    member this.Marka = marka
    member this.Model = model
    member this.ModelYılı = modelYılı

let hondaCrv = Araba(marka="Honda",model="CRV",modelYılı=2017)
let hondaHrv:Araba = null
```

Örnekte **null** değerlere **[<AllowNullLiteral>]** ile dekore ederek izin verdiğimiz **Araba** isimli bir sınıf tanımlanıyor. Bu sınıftan oluşturulan ilk değer (hondCrv) normak bir değer, ikinci değer (hondaHrv) ise null.

null değerlere izin verilen F# tiplerinde veya .NET standard kütüphanesindeki fonksiyonların **null** olabilen dönüş değerleri için **null** değer kontrolü F#'da normal tip ve değerler için yapılan kontroller ile aynı şekilde yapılır.

```
(* 03_5_01_null*)

let markayıGetir (a:Araba) : string =
    if a = null then
        "Geçerli bir araba örneği verilmemiş!"
    else
        a.Marka


markayıGetir hondaCrv
markayıGetir hondaHrv
```

Eğer fonksiyonunuza geçilecek paramerenin değerinin null olup olamayacağını net olarak bilmiyorsanız ve **null** için özel davranış kodlamanız gerekiyorsa F# standard kütüphanesindeki **box** fonksiyonunu veya F# 4 kullanıyorsanız **isNull** fonksiyonun aşağıdaki gibi kullanabilirsiniz.

```
(* 03_5_01_null*)

let nullMu değer = box değer = null
let nullMu' değer = isNull değer

nullMu hondaCrv
nullMu hondaHrv

nullMu' hondaCrv
nullMu' hondaHrv
```

Bilgi Kutusu (BİLGİ): box fonksiyonu F#'daki herhangi bir değeri referans tipinden bir nesneye dönüştürmek için kullanılır. Örneğin tam sayı değeri **1** basit bir değerdir, **box 1** ifadesi ile referans tipinden bir nesneye dönüştürüldü. Referans tipinden nesnelerin değeri **null** olabileceği için null kontrolünde box yöntemi kullanılabilir. **box** fonksiyonunun tersi işlem yapmak için de **unbox** fonksiyonu kullanılabilir.

Tuple (Değer Grubu)

Farklı tiplerde değerleri gruplamak için kullanılan tipe **değer grubu** (tuple) denir. F#'da değer gruplar virgül ile ayrılmış değerler şeklinde aşağıdaki formata uygun olarak tanımlanır.

```
      let değer_adı = (değer1,değer2,değer3)
```

Bilgi Kutusu (İPUCU): Değer grupları tanımlarken parantez kullanımı opsiyoneldir, ancak kod okunaklılığı açısından parentezleri kullanmanızı öneririm.

Değer gruplarının tipi **değer1_tipi * değer2_tipi * değer3_tipi** şeklinde yazılır.

Örneğin;

```
int * string * float
```

şeklindeki ifade ilk değeri **int**, ikinci değeri **string** üçüncü değeri **float** tipinden olan bir değer grubun tipini ifade eder.

```
(* 03_5_02.fsx *)

let yazar = ("Ali","Özgür",1979,9)
// Değer grubunun imzası şöyledir
// val yazar : string * string * int * int

// Değer tipleri tanımlı değer grubu
// Doğum günün kutlu olsun Ersel Özgür :)
let kardeş : string * string = ("Ersel","Özgür")
```

Değer grupları başka değer gruplarını da barındırabilir. Aşağıdaki örnekte **baba** isimli değer grubunun 3. elemanı yine bir değer grubudur.

```
let baba = ("Ali","Özgür", ("Arda","Özgür"))
```

Sadece iki değeri olan grupların elemanlarının değerlerini sökmek için **fst** ve **snd** standard kütüphane fonksiyonları kullanılabilir.

```
let baba = ("Ali","Özgür", ("Arda","Özgür"))

let çocuk = ("Arda","Özgür")

let çocukAd = fst çocuk
```

Değer grubundaki tüm elemanların değerleri tek bir satırda aşağıdaki gibi sökülerek ayrı ayrı ifadelere atanabilir. Değer grubunun bazı elemanları değer sökme işlemi sırasında göz ardı edilmek isteniyorsa da _ simgesini kullanılır.

```
// Tüm değerleri ayrı ayrı birer ifadeye atayalım
let babaAd,babaSoyad,doğumYılı,doğumAyı = yazar

// Bazı değerleri _ ile sökme sırasında görmezden gelelim
let kişiAd,kişiSoyad,_ = baba
let kişiAd',_,çocuğu = baba
```

Değer grubunun eleman sayısından daha az veya daha fazla eleman sökülmek istenirse derleyici hata verir.

```
let kişiAd,kişiSoyad = baba
// Derleyici aşağıdaki hatayı verir
(*
Error FS0001: Type mismatch. Expecting a
    'string * string'
but given a
    'string * string * (string * string)'
The tuples have differing lengths of 2 and 3
*)
```

Değer gruplarını fonksiyonlara girdi parametresi olarak geçip fonksiyonlardan da değer grubu döndürülebilir.

```
// Değer grubu parametresi alan fonksiyon
let topla (x,y) = x + y

topla (43,-1)

// Değer grubu parametresi alıp
// değer grubu döndüren fonksiyon
let topla' (x,y) =
    let t = x + y
    (t,sprintf "%d + %d = %d" x y t)

let toplam,metin = topla'(43,-1)
```

Değerlerinin tipi olmayan değer gruplarının imzası **'a * 'b** şeklindedir

```
let değerleriYazdır (x,y) =
    printfn "Değerler x=%A, y=%A" x y
// Fonksiyonun imzası şöyledir
// val değerleriYazdır : x:'a * y:'b -> unit

değerleriYazdır (baba,çocuk)
değerleriYazdır (42,0)
```

Bilgi Kutusu (DİKKAT!): Parametre olarak değer grubu alan fonksiyonların çağırılarında çok dikkatli olmalısınız, çünkü **f(1,2)** çağırısı ile **f 1 2** çağırıları eş çağırılar değildirler. İlk çağrı girdi olarak iki elemanlı bir değer grubu alırken ikinci çağırıdaki fonksiyon iki ayrı girdi parametresi alır.

F#'da fonksiyonlarınız tasarlarken girdi parametrelerini tanımlamak için mümkün ise değer gruplarını kullanmayın, çünkü **let f (a,b) = ...** şeklinde (a,b) değer grubunu girdi parametresi olarak alan bir fonksiyon için **kısmi uygulama** yapılamaz. Fonksiyon parametresi olarak değer grupları eğer değer grubu gerçekten bir şeyi modelliyorsa kullanılmalıdır.

İfadelere, fonksiyon girdi parametresi veya dönüş değerlerine değer grubu tiplerini tanımlamak için aşağıdaki şablona uygulmalıdır.

```
      (ifade1, ifade2): değerTipi1 * değerTipi2
```

```
// Değer tipleri tanımlı değer grubu parametresi
let çarp ( (x,y):int*int ) : int * string =
    let ç = x * y
    (ç,sprintf "%d * %d = %d" x y ç)

çarp (42,1)
```

Örneğimizdeki **çarp** fonksiyonuna bakarsanız **((x,y):int * int)** ile iki değeri de tam sayı olan bir değer grubunu parametre olarak aldığını **:int * string** ile de ilk elemanı tam sayı (int) ikinci elemanı metin (string) olan bir değer grubu döndürdüğünü anlarız.

ALIŞTIRMA-1

Aşağıdaki fonksiyon tanımlarını yaparak imzalarını karşılaştırın.

```
let topla(x,y) = x + y

let topla' x y = x + y
```

Eşitlik

İki değer grubu eleman sayısı, elemanlarını tipleri ve elemanlarının değerleri aynı ise birbirine eşittir. Elaman sayısı farklı olan veya eleman sayısı aynı olan ancak tipleri farklı olan değer grupları karşılaştırılamaz, derleyici hata verir.

```
(1,2) = (1,2) // true
(1,2) = (2,1) // false
```

```
(1,"Ali") = (1,"Arda") // false
(1,2,3) = (1,2) // Derleyici hata verir
(1,2) = ("1",2) // Derleyici hata verir
```

Option (Opsiyon)

Bazı durumlarda belirli bir değeri olmayan ifadeler veya belirli bir değer döndürmeyen fonksiyonlar yazılması gerekebilir. Bu tip durumlarda **option** (opsiyon) tipi kullanılabilir. Opsiyon tipinin iki olası değeri vardır; **None** değer olmadığını ifade eder, **Some('a)** ise 'a tipinden bir değeri ifade eder. **Some('a)** opsiyon değerini güvenli tipleme için asıl değeri çevreleyen bir kabuk tip olarak da düşünebiliriz.

```
(* 03_5_03.fsx *)

// ----- option tipinden değer tanımlama -----

let değer1 = Some(5)
let değer2 = None
let değer3 : int option = Some(5)

let değer4 : int option = None

let değer5 : (int list) option = None
```

Opsiyon tipi **option** olarak ifade edilir. Örneğin **int option** tam sayı opsyonu, **string option** metin opsiyonu ve **int list option** tam sayı listesi opsiyonu ifade eder.

Opsiyonların değer barındırıp barındırmadığı **Option** modülü içindeki **isSome** ve **isNone** fonksiyonları kullanarak kontrol edilebilir. Opsiyon bir değer barındırıyorsa, yani None değilse, çevrelediği asıl değer **Option** modülü içindeki **get** fonksiyonu ile sökülebilir. Değeri **None** olan bir opsiyon için **get** fonksiyonu çağırısı **ArgumentException** tipinden bir istisna fırlatılır.

```
(* 03_5_03.fsx *)

// ----- option tipinden parametre kullanımı -----

// Araba isimli kayıt tipi
type Araba = {Marka:string;Model:string}

// None kontrolü ve Option.get kullanımı
let arabaBilgisiniVer (a: Araba option):string =
    if a = None then
        "Araba nesnesi belirtilmemiş"
    else
        let araba = Option.get a
        sprintf "Marka = %s, Model = %s" araba.Marka araba.Model
```

```
// Option.isNone ve Option.get kullanımı
let arabaBilgisiniVer' (a: Araba option):string =
    if Option.isNone a then
        "Araba nesnesi belirtilmemiş"
    else
        let araba = Option.get a
        sprintf "Marka = %s, Model = %s" araba.Marka araba.Model


// Option.isSome ve Option.get kullanmı
let arabaBilgisiniVer'' (a: Araba option):string =
    if Option.isSome a then
        let araba = Option.get a
        sprintf "Marka = %s, Model = %s" araba.Marka araba.Model
    else
        "Araba nesnesi belirtilmemiş"

let araba1 = None
let araba2 = Some({Marka="Honda";Model="CRV"})

arabaBilgisiniVer araba1
arabaBilgisiniVer araba2

arabaBilgisiniVer' araba1
arabaBilgisiniVer' araba2

arabaBilgisiniVer'' araba1
arabaBilgisiniVer'' araba2
```

Opsiyonlar fonksiyonlardan **null** değeri kullanılmadan bazı koşullarda bir değer bazı koşullarda da ***hiç bir değer** döndürmemek için kullanılabilir.

Örneğin bölme işleminde 0'a bölme tanımsız bir işlemdir. **bölüm'** fonksiyonunun bölüm parametresi olan **y** için 0 değeri geçildiğinde fonksiyonun dönüş değeri **None** olur.

```
(* 03_5_03.fsx *)

// ----- option tipinden değer döndürme -----

let bölüm (x:float) (y:float) = x / y

bölüm 5.0 0.0 // Sonuç : infinity
bölüm 5.0 2.0 // Sonuç : 2.5

let bölüm' (x:float) (y:float) =
    match y with
    | 0.0 -> None
    | _ -> Some(x / y)

bölüm' 5.0 0.0 // Sonuç : None

bölüm' 5.0 2.0 // Sonuç : Some(2.5)
```

Eşitlik

Opsiyon değerleri çevreledikleri tipin değeri eşit ise birbirine eşittir. Aynı tipten değerleri çevreleyen **Some** değerleri eşitlik için kontrol edilebilir, farklı tipleri çevreleyen **Some** değerlerinin eşitlik kontrolüne derleyici izin vermez.

```
(* 03_5_03.fsx *)

// ----- option eşitliği -----

None = None // true
Some(1) = Some(1) // true
Some(1) = Some(2) // false
//Some(1) = Some("1") // Derleyici hatası, çevrelenen tipler farklı
```

3.6 Yapısal Eşitlik

F#'da değer grupları (tuple), listeler (list), opsiyonlar (option), diziler (array), kayıtlar (record), bileşimler (union) ve çatılar (struct) gibi tipler otomatik olarak **yapısal eşitlik** (structural equality) denilen yaklaşımı destekler. F#'daki yapısal eşitlik yaklaşımı C++, Java ve C# gibi dillerdeki eşitlik yaklaşımından farklıdır, çünkü bu dillerde nesnelerin işaretçi eşitliği ile içerik eşitliği birbirinden farklı olarak ele alınır ve karşılaştırma kodunu sizin yazmanız beklenir. F#'da ise temel tiplerin hepsi için herhangi bir kodlama yapılmadan içerik eşitliği otomatik olarak tespit edilebilir.

Yapısal eşitliğin ne olduğunu daha iyi anlamak için önce C#'da referans eşitliği ve içerik eşitliği kontrolünün nasıl yapıldığına bakalım. İlk C# örneğimizde **Kişi** isimli bir sınıf tanımlıyoruz. Bu sınıfın **Ad** ve **Soyad** isimli iki alanı var. Main metodu içinde **Ad** ve **Soyad** alanlarının değeri aynı olan **Kişi** tipinden **kişi1** ve **kişi2** isimli iki değişken tanımlıyoruz. İlave olarak **kişi3** isimli bir değişken tanımlayıp değrini **kişi1** değişkeni olarak atıyoruz.

```
public class Kişi
{
    public string Ad{get;set;}
    public string Soyad{get;set;}
}

void Main()
{
    var kişi1 = new Kişi{Ad = "Ali", Soyad = "Özgür"};
    var kişi2 = new Kişi{Ad = "Ali", Soyad = "Özgür"};
    var kişi3 = kişi1;

    Console.WriteLine("kişi1 == kişi2 : {0}", kişi1 == kişi2);
    // Çıktı : "kişi1 == kişi2 : False"

    Console.WriteLine("kişi1 == kişi3 : {0}", kişi1 == kişi3);
    // Çıktı : "kişi1 == kişi3 : True"
}
```

C#'da == operatörü karşılaştırma için kullanılır ve sınıflar için varsayılan olarak işaretçi referanslarını yani değişkenlerin bellekteki adreslerini karşılaştırır. Bu nedenle içerikleri eşit olan ancak bellek adresleri farklı olan kişi1 ve kişi2 değişkenleri **kişi1 == kişi2** şeklinde karşılaştırılınca sonuç **False** olurken aynı bellek adresini (aynı nesneyi) işaret eden kişi1 ve kişi3 değişkenleri **kişi1 == kişi3** şeklinde karşılaştırıldığında sonuç **True** olur.

Özetle, C# ve benzeri dillerde aynı sınıftan nesnelerin içeriği otomatik olarak karşılaştırılabilir değildir. C#'da içerik karşılaştırması yapabilmek için **Kişi** sınıfının atası olan **System.Object**'den devraldığı **Equals** metodunu karşılaştırma mantığımıza uygun olarak kodlamamız gerekiyor. Ancak, **Equals** metodunu kodlasak bile hala **==** operatörünü karşılaştırma için kullanamayız bu nedenle içerik karşılaştırmasını **Equals** metodunu kullanarak yapmamız gerekir.

```
public class Kişi
{
    public string Ad{get;set;}
    public string Soyad{get;set;}

    /*
        Equals Sytem.Object tipinin metodu
        C#'da System.Object tüm sınıfların atası, o nedenle
        Kişi sınıfında da Eeuals metodunu override ederek kendi
        kodumuzu yazabiliriz
    */
    public override bool Equals(object kişi)
    {
        var k = kişi as Kişi;
        return k == null
            ? false
            : this.Ad == k.Ad && this.Soyad == k.Soyad;
    }
}

void Main()
{
    var kişi1 = new Kişi{Ad = "Ali", Soyad = "Özgür"};
    var kişi2 = new Kişi{Ad = "Ali", Soyad = "Özgür"};
    var kişi3 = kişi1;

    // İçerik eşitliği
    Console.WriteLine("kişi1 == kişi2 : {0}", kişi1.Equals(kişi2));
    // Çıktı : "kişi1 == kişi2 : True"

    Console.WriteLine("kişi1 == kişi3 : {0}", kişi1.Equals(kişi3));
    // Çıktı : "kişi1 == kişi3 : True"

    // İşaretçi referansı eşitliği
    Console.WriteLine("kişi1 == kişi2 : {0}", kişi1 == kişi2);
    // Çıktı : "kişi1 == kişi2 : False"

    Console.WriteLine("kişi1 == kişi3 : {0}", kişi1 == kişi3);
    // Çıktı : "kişi1 == kişi3 : True"
}
```

Bilgi Kutusu (BİLGİ): Kitabımız C# kitabı olmadığı için C#'daki eşitlik ve karşılatırma yöntemlerinin hepsinden bahsetmiyoruz. **Equals** metodu ile sağlanan eşitlik kontrolü imkanı basit yöntemlerden sadece birisidir, bunun dışında C#'da == operatörünün yeniden kodlanması, farklı kütüphane sınıflarına özel eşitlik ve karşılaştırma işlemlerinin kodlanması gibi yöntemler de uygulanabilir.

C# örneğimizi F# ile tekrarladığımızda içerik eşitliği kontrollerini ilave kodlama yapmadan otomatik olarak sağlanan **yapısal eşitlik** sayesinde yapılabildiğini görüyoruz.

```
(* 03_6_03.fsx *)

// Kişi isimli kayıt tipi
type Kişi = {Ad:string;Soyad:string}

// kişi1, kişi2 ve kişi3 değerler
let kişi1 = {Ad="Ali";Soyad="Özgür"}
let kişi2 = {Ad="Ali";Soyad="Özgür"}
let kişi3 = kişi1

let kişi4 = {Ad="Arda";Soyad="Özgür"}


printfn "kişi1 = kişi2 : %b" (kişi1 = kişi2)
// Çıktı : kişi1 = kişi2 : true

printfn "kişi1 = kişi3 : %b" (kişi1 = kişi3)
// Çıktı : kişi1 = kişi3 : true

printfn "kişi2 = kişi3 : %b" (kişi2 = kişi3)
// Çıktı : kişi2 = kişi3 : true

printfn "kişi1 = kişi4 : %b" (kişi1 = kişi4)
// Çıktı : kişi1 = kişi4 : false
```

Yapısal eşitlik ile sadece eşitlik/farklılık kontrolü değil aynı zamanda büyüklük/küçüklük karşılaştırması da otomatik olarak yapılabilir. Büyüklük/küçüklük karşılaştırması yapılırken öncelikle aşağıdaki koşulların sağlanması gerekir.

- Liste ve dizi gibi tiplerin eleman tipleri ve eleman sayısı aynı olmalı.
- Değer gruplarının değer tipleri ve sayıları aynı olmalı. Örneğin: **int * string * float** ile **int * int * float** tipinden değerler içeren gruplar karşılaştırılamaz.
- Some('a) şeklinde tanımlanan opsiyonların çevreledikleri '**a** tipi aynı olmalı. Örneğin **Some(1)** ile **Some("1")** karşılaştırılmaz.
- Kayıt (record), çatı (struct) ve bileşimlerin (union) karşılaştırılabilmesi için karşılaştırılan değerlerin tipleri aynı olmalıdır.

Bu koşullar derleyici tarafından kontrol edilir, eğer bir uyumsuzluk varsa derleyici hata üretir. Bu koşullar sağlandığında ise her bir tip için aşağıdaki kurallara göre büyüklük/küçüklük karşılaştırması yapılır.

- Liste ve dizi gibi tiplerde aynı pozisyondaki elemanlar karşılaştırılarak büyüklük/küçüklük kararı verilir. İlk farklı olan pozisyondaki eleman değerlerine göre karar verilir. Örneğin **let a = [1;2;3]** listesi ile **let b = [1;3;3]** listesi için **a > b** karşılaştırması yapılırken değerleri eşit olmayan ilk elemanları 2. elemanlardır . Bu nedenle, **a > b** karşılaştırması **a.[1] > b.[2]** olarak yorumlanır ve sonucu **false**'dır. Örneğimizde bu karşılaştırmanın sonucu **false** olacaktır.
- Değer grupları için de diziler ve listelerdeki gibi grubu oluşturan değerler pozisyon pozisyon karşılaştırılır. Örneğin **let a = (1,2,3)** değer grubu ile **let b = (1,3,3)** değer grubu için **a > b** karşılaştırması yapılırken değerleri eşit olmayan ilk değerler 2. elemanlardır. Bu nedenle **a > b** karşılaştırması **2 > 3** olarak yorumlanır ve sonucu **false**'dır.
- Some('a) şeklinde tanımlanan opsiyonlar için çevreledikleri değerler karşılaştırılır. Örneğin **let a = Some(1)** ve **let b = Some(2)** için **a > b** karşılaştırması **1 > 2** olarak yorumlanır ve sonucu **false**'dır.
- Kayıt (record) ve çatı (struct) tiplerinde karşılaştırma alan alan yapılır. İlk farklı olan alanın değerine göre de karar verilir. Örneğin **let a = {Ad="Ali";Yaş=37}** kaydı ile **let b = {Ad="Ali";Yaş=45}** kaydı için **a > b** karşılaştırması yapılırken değerleri eşit olmayan ilk alan **Yaş** alanıdır. Bu nedenle,**a > b** karşılaştırması **37 > 45** olarak yorumlanır ve ve sonucu **false**'dır.
- Bileşimler için karşılaştırma yapılırken ise bileşim değerinin bileşim tip tanımındaki sırası dikkate alınır. Örneğin **type Şehir = Adana|Bursa|İstanbul** şeklindeki bir bileşim tanımına göre **İstanbul > Adana** karşılaştırmasının sonucu **true**, **Adana > Bursa** karşılaştırmasının sonucu ise **false** olur.

```
(* 03_6_04.fsx *)

// ---- Değer Grupları (tuple) Eşitliği ve Karşılaştırma ----
let değerGrubu1 = (1,2)
let değerGrubu2 = (1,2)
let değerGrubu3 = (2,1)
let değerGrubu4 = (2,3)

değerGrubu1 = değerGrubu2 // true
değerGrubu1 = değerGrubu3 // false

değerGrubu1 < değerGrubu3 // true
değerGrubu4 > değerGrubu3 // true


// ---- Liste (list) Eşitliği ve Karşılaştırma ----
let liste1 = [1..5]
let liste2 = [1..5]
let liste3 = [
    for i in 1..5 do
        if i = 1 then
            yield 2
        else if i = 2 then
            yield 1
        else
            yield i
]

let liste4 = [
    for i in 1..5 do
        if i = 2 then
            yield 0
        else
            yield i
]

liste1 = liste2 // true
liste1 = liste3 // false
liste3 > liste1 // true
liste4 > liste1 // false
```

```
// ---- Dizi (Array) Eşitliği ve Karşılaştırma ----
let dizi1 = [1..5]
let dizi2 = [1..5]
let dizi3 = [
    for i in 1..5 do
        if i = 1 then
            yield 2
        else if i = 2 then
            yield 1
        else
            yield i
]

let dizi4 = [
    for i in 1..5 do
        if i = 2 then
            yield 0
        else
            yield i
]


dizi1 = dizi2 // true
dizi1 = dizi3 // false
dizi3 > dizi1 // true
dizi4 > dizi1 // false


// ---- Opsiyon (option)  Eşitliği ve Karşılaştırma ----
let opsiyon1 = Some(1)
let opsiyon2 = Some(1)
let opsiyon3 = Some(2)

opsiyon1 = opsiyon2 // true
opsiyon1 = opsiyon3 // false
opsiyon3 > opsiyon1 // true
```

```
// ---- Kayıt (Record) Eşitliği ve Karşılaştırma ----
type Kişi = {Ad:string;Soyad:string;DoğumYılı:int}

// kişi1, kişi2 ve kişi3 değerler
let kişi1 = {Ad="Ali";Soyad="Özgür";DoğumYılı=1979}
let kişi2 = {Ad="Ali";Soyad="Özgür";DoğumYılı=1979}
let kişi3 = {Ad="Ali";Soyad="Özgür";DoğumYılı=1980}
let kişi4 = {Ad="Arda";Soyad="Özgür";DoğumYılı=1979}

kişi1 = kişi2 // true
kişi1 = kişi3 // false

kişi1 < kişi3 // true
kişi1 < kişi4 // true

// ---- Çatı (struct)  Eşitliği ve Karşılaştırma ----

type Nokta =
    struct
        val x: float
        val y: float
        new(x,y) = {x=x;y=y}
    end

let nokta1 = Nokta(1.0,0.0)
let nokta2 = Nokta(1.0,0.0)

let nokta3 = Nokta(1.0,2.0)
let nokta4 = Nokta(2.0,0.0)

let nokta5 = Nokta(0.0,2.0)

nokta1 = nokta2 // true
nokta1 = nokta3 // false
nokta1 = nokta4 // false
nokta3 > nokta1 // true
nokta4 > nokta1 // true
nokta5 > nokta1 //false
```

```
// ---- Bileşimler (union)  Eşitliği ve Karşılaştırma ----

type Şehir = Adana|Bursa|İstanbul

let bursa = Bursa
let bursa' = Bursa
let adana = Adana
let istanbul = İstanbul

bursa = bursa' // true
bursa = adana // false
bursa = istanbul // false
bursa > adana // true
adana > istanbul // false
```

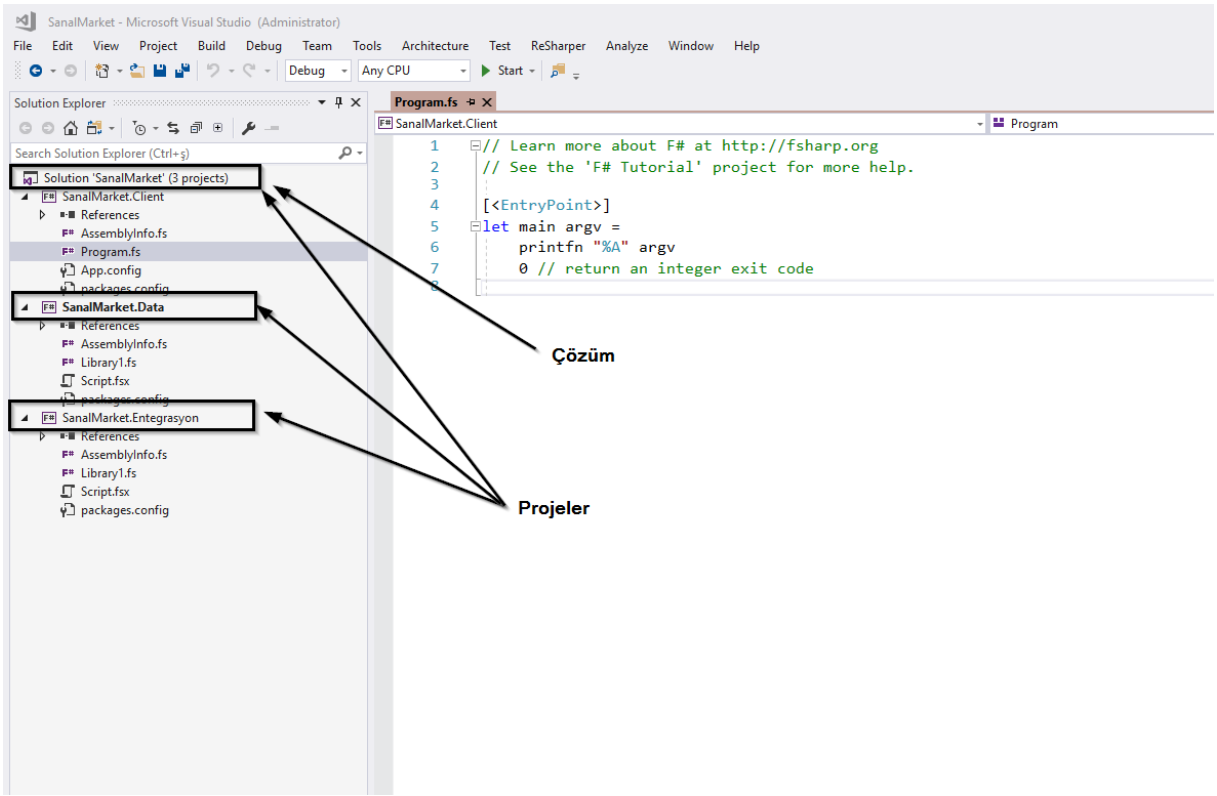
3.7 Kod Organizasyonu

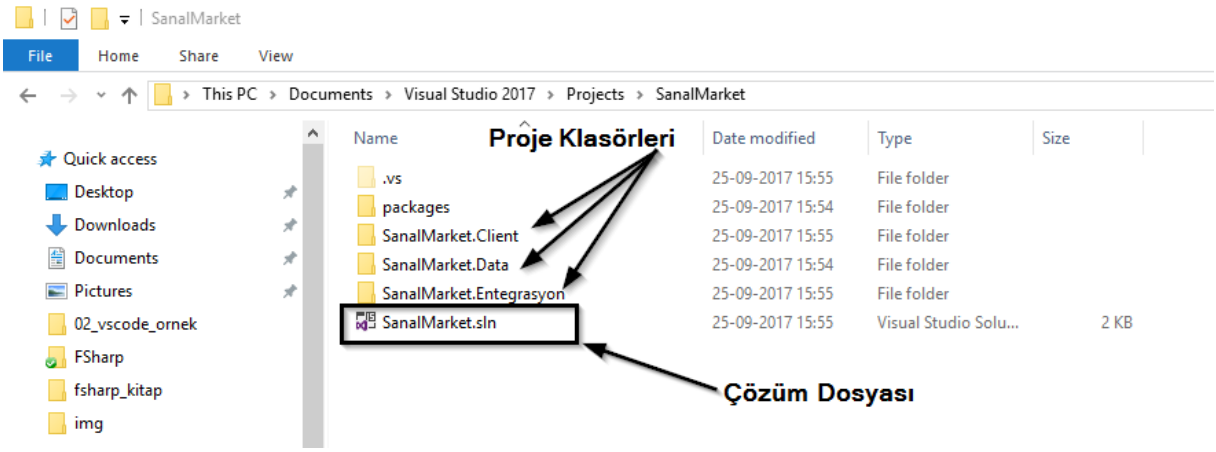
Çözümler ve Projeler

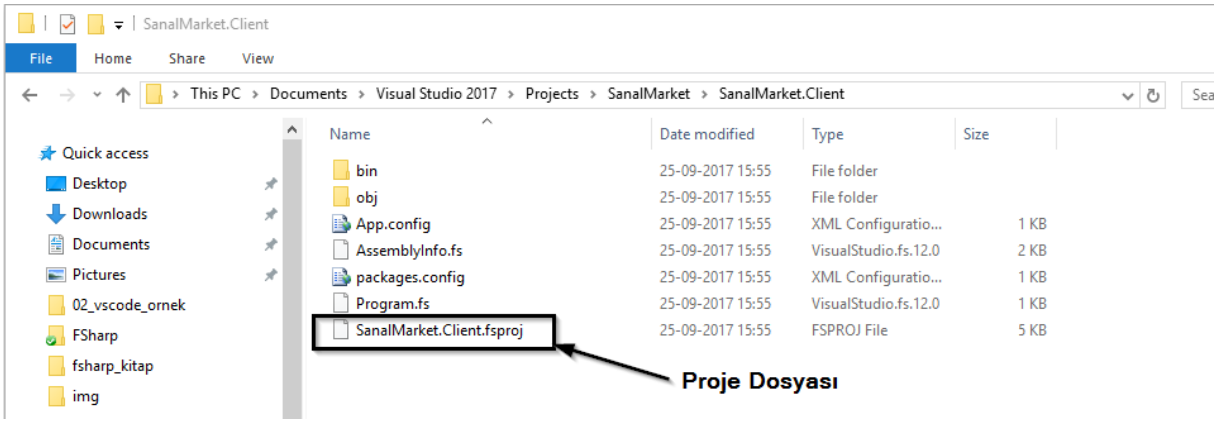
Büyük ve karmaşık uygulamalar geliştirmek için uygulamanın her bir işlevsel parçasını farklı birer büyüklük olarak ele alıp bunların hepsini de bir çatı altında birleştirmek kaçınılmazdır. .NET platformunda bu seviyedeki kod organizasyonu projeler ve çözümler (solution) kullanılarak yapılır. Uygulamanın tamamı bir çözüm (solution) olarak oluşturulup uygulamanın farklı işlevleri veya modüller de ayrı birer proje olarak bu çözüme eklenir.

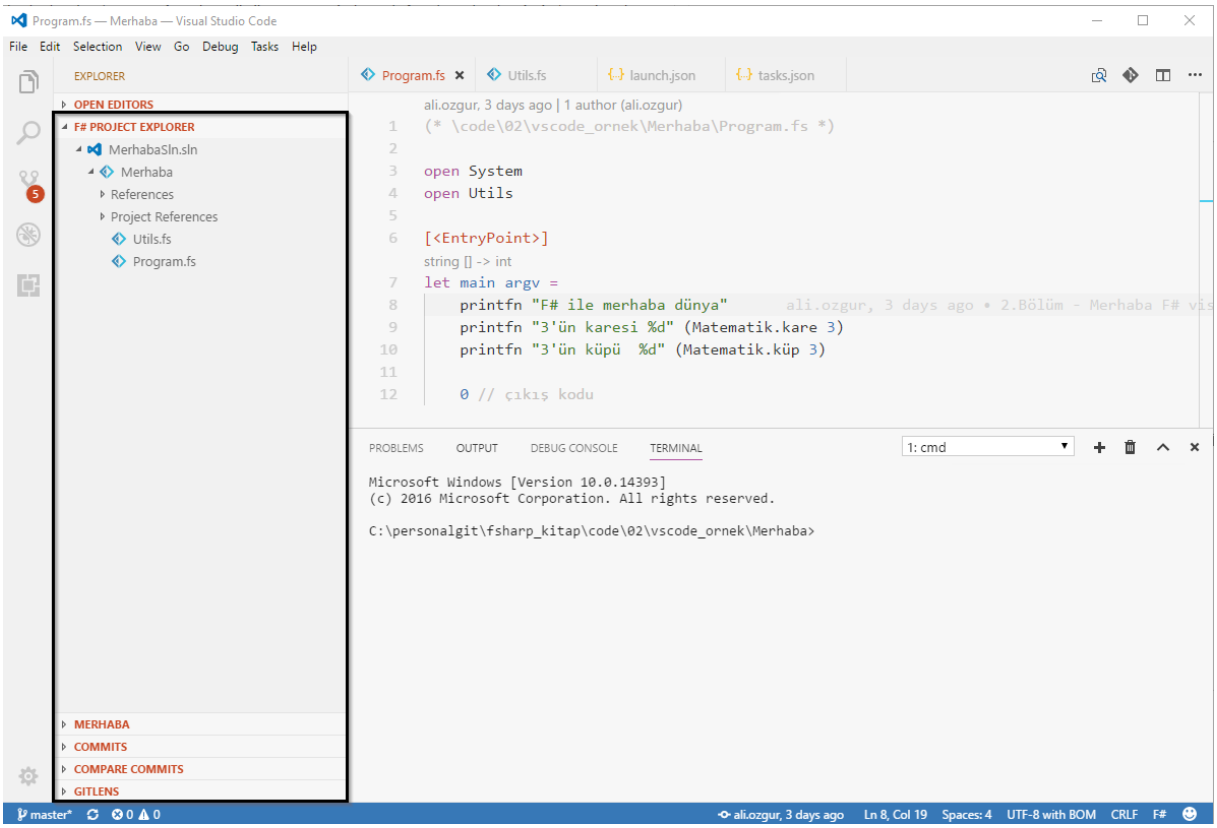
.NET ile veya F# özelinde uygulama geliştirilirken projelerin ve çözümlerim kullanılması zorunlu değildir. Bu iki yapı içinde ayarların ve kod dosyalarına ve diğer dosyalara linklerin olduğu birer dosyadan ibarettir. Çözüm dosyaları **sln** uzantılı, proje dosyaları ise **fsroj** uzantılı içinde XML notasyonuna uygun bilgilerin yer aldığı dosyalardır. Bu iki dosyanın içeriğini elle düzenlemeniz beklenmez, çünkü F# destekleyen tüm editörler kullanıcı arayüzünden erişilebilen komutlar ile çözümlere proje ekleme, projelere dosya ekleme, dosya sıralamasını değiştirme ve kütüphanelere referans ekleme gibi işlemleri destekler.

Çözümler ve projelere uygulama kodunun organize edilmesinin yanı sıra derleyicinin tüm dosyaları doğru sırada derlemesini ve doğru çıktıyı üretmesini de kolaylaştırır. Ancak, çözüm ve projeler doğrudan derleyici tarafından kullanılmaz, dolaylı olarak F# destekleyen kod editörleri bu dosyaların içeriğini okuyup yorumlayarak derleyiciye doğru komutların gönderilmesini sağlar.




SanalMarket
File
Home
Share
View
This PC › Documents › Visual Studio 2017 › Projects › SanalMarket
Quick access
Desktop
Downloads
Documents
Pictures
02_vscode_ornek
FSharp
fsharp_kitap
img
Name
Date modified
Type
Size
Proje Klasörleri
.vs
packages
SanalMarket.Client
SanalMarket.Data
SanalMarket.Entegrasyon
SanalMarket.sln
25-09-2017 15:55
25-09-2017 15:54
File folder
Visual Studio Solu...
2 KB
Çözüm Dosyası


SanalMarket.Client
File
Home
Share
View
This PC › Documents › Visual Studio 2017 › Projects › SanalMarket › SanalMarket.Client
Quick access
Desktop
Downloads
Documents
Pictures
02_vscode_ornek
FSharp
fsharp_kitap
img
Name
Date modified
Type
Size
bin
obj
App.config
AssemblyInfo.fs
packages.config
Program.fs
SanalMarket.Client.fsproj
25-09-2017 15:55
File folder
XML Configuratio...
VisualStudio.fs.12.0
FSPROJ File
1 KB
2 KB
5 KB
Proje Dosyası


Program.fs — Merhaba — Visual Studio Code
File Edit Selection View Go Debug Tasks Help
EXPLORER
OPEN EDITORS
F# PROJECT EXPLORER
MerhabaSln.sln
Merhaba
References
Project References
Utils.fs
Program.fs
MERHABA
COMMITS
COMPARE COMMITS
GITLENS
Program.fs
Utils.fs
launch.json
tasks.json
ali.ozgur, 3 days ago | 1 author (ali.ozgur)
(* \code\02\vscode_ornek\Merhaba\Program.fs *)
open System
open Utils
[<EntryPoint>]
string [] -> int
let main argv =
printfn "F# ile merhaba dünya"
printfn "3'ün karesi %d" (Matematik.kare 3)
printfn "3'ün küpü %d" (Matematik.küp 3)
0 // çıkış kodu
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
1: cmd
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.
C:\personalgit\fsharp_kitap\code\02\vscode_ornek\Merhaba>
master*
ali.ozgur, 3 days ago
Ln 8, Col 19
Spaces: 4
UTF-8 with BOM
CRLF
F#

Bu bölümde 03_08.png nolu referans resim kullanılacak.
Bu bölümde 03_09.png nolu referans resim kullanılacak.
Bu bölümde 03_10.png nolu referans resim kullanılacak.
Bu bölümde 03_11.png nolu referans resim kullanılacak.

Dosya Sıralaması

F# derleyicisi kod dosyalarını yukarıdan aşağıya, sağdan sola çözümler. F#'da herhangi bir ifade, fonksiyon veya tip tanımının kullanılabilmesi için kullanılacağı yerden önce tanımlanmış olması ve F# derleyicisi tarafından çözümlenmiş olması gerekiyor. C++, Java veya C# gibi dillerde fonksiyon veya tip tanımlarının hangi aşamada yapıldığını fazla önemi yoktur. Örneğin C# ile aşağıdaki gibi bir tanımlama ve sıralama geçerli bir kullanımdır.

```
using System;

public class Program
{

    public static void Main()
    {
        var kişi = new Kişi{Ad="Arda",Soyad="Özgür"};
        Console.WriteLine($"Kişi bilgisi : {kişi}");

        var test = TestMetodu();
        Console.WriteLine($"Test metod sonucu : {test}");

    }

    public static string TestMetodu()
    {
        return "Bu bir test metodudur.";
    }
}

public class Kişi
{
    public string Ad{get;set;}
    public string Soyad{get;set;}

    public override string ToString()
    {
        return $"{this.Ad} {this.Soyad}";
    }
}
```

C# örneğinde **Kişi** sınıfı kullanıldığı **Program** sınıfının **Main** metodundan sonra tanımlanmıştır. Buna rağmen C# derleyicis hata vermez ve program çalışır. Benzer bir sıralamayı F# için yapmaya çalıştığımızda ise F# derleyicisi hata verecektir.

```
(* 03_7_01.fsx *)
// Kişi tipinden değer
let kişi = {Ad="Arda";Soyad="Özgür"}

// kişi değerini ekrana yazdırma
printfn "Kişi bilgisi : %s" (kişiBilgisi kişi)

// Kişi bilgisi fonksiyonu tanımı
let kişiBilgisi (k:Kişi) =
    sprintf "%s %s" k.Ad k.Soyad


// Kişi kayıt tipi tanımı
type Kişi = {Ad:string;Soyad:string}
```

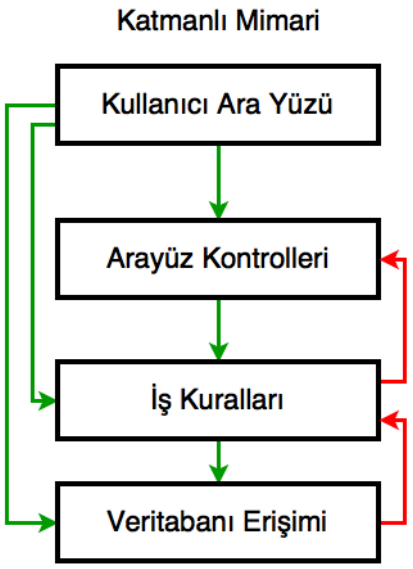
Yukarıdaki F# kod parçasında **Kişi** tipi ve **kişiBilgisi** fonksiyonları kullanıldıkları noktadan sonra tanımlandığı için F# derleyicis bu kod parçasını derleyemez. Örnek kod parçasının **Kişi** tipi önce sonra da **kişiBilgisi** fonksiyonu gelecek şekilde değer, tip ve fonksiyon tanımları kullanılmadan önce tanımlanacak şekilde düzeltilmesi gerekir.

```
(* 03_7_01.1.fsx *)

// Kişi kayıt tipi tanımı
type Kişi = {Ad:string;Soyad:string}

// Kişi bilgisi fonksiyonu tanımı
let kişiBilgisi (k:Kişi) =
    sprintf "%s %s" k.Ad k.Soyad

// Kişi tipinden değer
let kişi = {Ad="Arda";Soyad="Özgür"}

// kişi değerini ekrana yazdırma
printfn "Kişi bilgisi : %s" (kişiBilgisi kişi)
```

Tek bir kod dosyası içindeki bu sıralama benzer şekilde birden fazla kod dosyasından oluşan programlar için de önemlidir. **Kişi** kayıt tipini **Kisi.fs** isimli bir dosyada **kişiBilgisi** fonksiyonunu da **Fonksiyonlar.fs** isimli bir dosyada tanımlayıp bunları da **Program.fs** isimli bir dosyada kullansaydınız dosyaların derlenmes sırası önce **Kişi.fs** sonra **Fonksiyonlar.fs** ve en son **Program.fs** şeklinde olmalıydı.

F#'da dosya ve tanımlama sırasının ciddi bir kural olarak derleyici tarafından denetlenmesinin en önemli nedeni **dairesel bağımlılık** (cyclic dependency) durumunun oluşmasının engellenmesidir. Özellikle katmanlı mimari (layered architecture) modele göre geliştirilen sistemlerde hiyerarşinin

üstün seviyelerindeki katmanların prensip olarak sadece hiyerarşinin alt seviyelerindeki katmanlara bağımlı olması gerekir, tersinin olması istenmez. Ancak bazı programlama dillerinde hiyerarşinin alt seviyesindeki katmanların karşılıklı olarak üst seviyedeki katmanlara bağımlı olmasını engellemek için herhangi bir kural uygulanmaz. BU tür dairesel bağımlılıklar katmanlı mimarinin maksadını aşan yöntemler ile kırılmasına neden olur.



Bu bölümde 03_12.png nolu referans resim kullanılacak.

Yukarıdaki örnek katmanlı mimari diyagramında **Veritabanı Erişimi** katmanının **İş Kuralları** katmanına bağımlı olması ve **İş Kuralları** katmanının da **Arayüz Kontrolleri** katmanına bağımlı olması istenmeyen dairesel bağımlılığa örnektir.

Bilgi Kutusu (BİLGİ): Programlama dillerinin bazılarında dairesel bağımlılık için bir engel olmamasına rağmen kod editörlerinin bir çoğu, örneğin Visual Studio'da C# için, destekledikleri dillerin yapısına göre bu bağımlılıkların oluşturulmasını engelleyecek veya en azında uyarılar üretecek araçlara sahiptir.

Dosyaların kendi aralarında ve bir dosyanın içindeki değerlerin, tiplerin ve fonksiyonların sırasının önemli olması programınızın tasarımını yaparken farklı katmanlar arasındaki hiyerarşiyi ve iletişim yöntemini daha ayrıntılı düşünmenizi sağlar.

F# projelerinde dosyaların sırası **fsproj** uzantılı proje dosyalarında kayıt altında tutulur ve derleyicinin hangi dosyayı önce hangisini sonra derleyeceği bu dosyaya göre belirlenir. Visual Studio, Visual Studio Code, Visual Studio for Mac ve JetBrains Rider gibi F#'ı destekleyen editörler dosya sıralarını kolayca düzenlemeniz için kısayol komutları sunar. Bu editörleri kullanmıyorsanız **fsc** veya **fsharpc** komutunu çalıştırırken kod dosyalarını sıralı olarak vermeniz gerektiğini de unutmayın.

Bilgi Kutusu (İPUCU): Kodunuzu **fsc** veya **fsharpc** ile komut satırından manuel olarak derliyorsanız dosya adlarının alfabetik olarak sıralanması esasına dayalı basit shell scritler ile (build script) derleyici komutunu oluşturup derleyiciyi çalıştırabilirsiniz.

Modüller ve Alan Adları

Modüller

Programlarınızda değerler, fonksiyonlar ve tipler temel modelleme ve organizasyon yapılarıdır. Bu yapıların bir üst seviyesinde programların katmanları ve farklı görevleri yerine getiren parçaları modülleri kullanılarak organize edilir.

F#'da modüller iki seviyede tanımlanır

- Üst seviye modüller (top level modules)
- İç modüller (nested modules)

Programların kodunu organize etmek için F# kod dosyaları üst seviye modül tanımı ile başlamalı. Üst seviye modüller aşağıdaki format uygun olarak dosyanın başında ve sol tarafında herhangi bir girinti bırakılmadan tanımlanır.

```
      module [Modül Adı]
```

Üst seviye modüllerin içinde yer alan kod blokları da herhangi bir girinti bırakılmadan sola yanaşık bir şekilde tanımlanır.

```
(* 03_7_02.fs *)
module SanalMarket
let MarkaAdı = "Sanal Market"
let Echo x =
    sprintf "%A" x

type Müşteri = {Ad:string;Soyad:string}
```

Yukarıdaki örneğimizde **03_7_02.fs** kod dosyasının içinde **SanalMarket** isimli üst seviyede bir modül tanımladık. Bu modülün içinde **MarkaAdı** isimli bir değer, **Echo** isimli bir fonkisyon ve **Müşteri** isimli bir kayıt tipi oluşturduk. SanalMarket modülü içinde tanımlı bu ifadeleri **03_7_03.fsx** script dosyası içinden aşağıdaki gibi kullanabiliriz.

```
(* 03_7_03.fsx *)
#load "03_7_02.fs"

// SanalMarket modülünü erişime açıyoruz
open SanalMarket

printfn "Marka Adı = %s" MarkaAdı
Echo "Sanal Market Client"

let müşteri = {Ad="Mahmut";Soyad="Tuncer"}
```

- **#load** FSI direktifi ile modülün bulunduğu dosya ortama yüklenir.
- **open SanalMarket** ifadesi ile modül açılır ve içeriği kullanılabilir hale gelir.

Modül içeriğine erişmek için **open** komutu ile modül açılmaz ise modülün içindeki değerler, fonksiyonlar ve tipler tam isimleri ile çağırılmalıdır.

```
(* 03_7_03.fsx *)
#load "03_7_02.fs"

//open SanalMarket

printfn "Marka Adı = %s" SanalMarket.MarkaAdı
SanalMarket.Echo "Sanal Market Client"

let müşteri = {
    SanalMarket.Müşteri.Ad="Mahmut"
    SanalMarket.Müşteri.Soyad="Tuncer"}
```

Modül içindeki ifadelerin tam isimlerinin formatı aşağıdaki gibidir

```
      üst-modül-adı.alt-modül-adı.[Değer | Fonksiyon | Tip]
```

Tam isim formatında üst modül adından sonra iç modül adları gerekli sayıda nokta ile ayrılmış olarak yazılabilir.

Bilgi Kutusu (DİKKAT): Örneklerimizdeki **#load** direktifi sadece FSI ile çalışıyorsak kullanılabilir. Bir editör içinde geliştirme yapıyorsanız F# kodunu #load ile yüklemenize gerek olmadan sadece **open** ile modül işlevlerine erişim sağlayabilirsiniz.

Üst seviye modüllerin altında kodun organizasyonu açısından gerekli ise iç içe ilave modüller de tanımlanabilir. İç içe modüller üst seviye modülün altında herhangi bir girinti bırakılmadan **module =** formatına uygun olarak yazılır. Üst seviye modüller ile iç modüllerin oluşturulması arasındaki tek fark iç modüller oluşturulurken modül adından sonra = operatörünün kullanılmasıdır. İç modüllerin altındaki diğer iç modüller de yine aynı formata uygun olarak fakat seviyesine göre uygun miktarda girintiler bırakılarak tanımlanır.

Yukarıdaki örneğimizi **SanalMarket** üst modülü altında **Sepet** isimli bir iç modül ve bunun altında da **Utils** isimli başka bir iç modül olacak şekilde geliştirelim.

```
(* 03_7_02.fs *)
module SanalMarket
let MarkaAdı = "Sanal Market"
let Echo x =
    sprintf "%A" x

type Müşteri = {Ad:string;Soyad:string}

// SanalMarket modülü altında alt seviye modül
module Sepet =
    type Ürün={Ad:string;Fiyat:decimal}
    type Sepet = {Müşteri:Müşteri; Ürünler: Ürün list}

    // Sepet alt modülü altında başka bir alt modül
    module Utils =
        let ürünOluştur ad fiyat =
            {Ad="iPhone X";Fiyat=fiyat}
        let sepetOluştur ad soyad ürünler =
                {Müşteri={Ad=ad;Soyad=soyad}; Ürünler= ürünler}
```

Bu iki iç modülün işlevlerini **SanalMarket.Sepet** iç modülünü ve bunun altındaki **SanalMarket.Sepet.Utils** iç modülünü **open** ile açarak aşağıdaki gibi kullanırız.

```
(* 03_7_03.fsx *)

//SanalMarket üst modülü altındaki
//  Sepet iç modülü
//erişime açıyoruz
open SanalMarket.Sepet
let iPhone7 = {Ad="iPhone 7";Fiyat=5099M}

//SanalMarket üst modülü altındaki
//  Sepet iç modülünün altındaki
//      Utils iç modülünü
//erişime açıyoruz
open SanalMarket.Sepet.Utils

let iPhoneX = ürünOluştur "iPhone X" 6099M

// Değer kavrama ile ürünleri oluşturup
// listeyi |> ile sepetOluştur fonksiyonuna aktarıyoruz

[
    for i in 3..6 do
        yield {Ad= sprintf "iPhone %d" i;Fiyat= decimal(i) * 1000M}
] |> sepetOluştur "Mahmut" "Tuncer"
```

Bilgi Kutusu (BİLGİ): FSI ile etkileşimli olarak çalıştırılan kod parçalarında modül tanımı yapılmasa bile F# varsayılan olarak yazılan kodu dosya adı ile aynı isimde bir modül altında derler ve yorumlar.

Alan Adları

F#'da üst seviye modüller yerine alan adları da **namespace alan-adı** formatına uygun olarak tanımlanabilir. Alan adları dosyanın tepesinde ve soldan hiç bir girinti verilmeden tanımlanır. Alan adlarının altında iç içe modül tanımları yapılabilir ancak iç içe alan adı tanımlanması mümkün değildir.

Alan adlarının altında üst seviyedeki modüllerden farklı olarak sadece tip tanımı yapılabilir, alan adları altında doğrudan fonksiyon tanımı yapılamaz veya do

Örneklerimizdeki **SanalSepet** üst seviye modülünü aşağıdaki gibi alan adı kullanacak şekilde düzenleyebiliriz. Ancak, dikkat ederseniz üst seviye modül içinde tanımlı olan **MarkaAdı** değerini ve **Echo** fonksiyonunu alan adı altındaki **Genel** isimli bir iç modüle taşımak zorunda kaldık. **Müşteri** tip tanımını ise alan adı altında bırakabilidik.

```
(* 03_7_02.1.fs *)

namespace SanalMarket

type Müşteri = {Ad:string;Soyad:string}

module Genel =
    let MarkaAdı = "Sanal Market"
    let Echo x =
        sprintf "%A" x

// SanalMarket modülü altında alt seviye modül
module Sepet =
    type Ürün={Ad:string;Fiyat:decimal}
    type Sepet = {Müşteri:Müşteri; Ürünler: Ürün list}

    // Sepet alt modülü altında başka bir alt modül
    module Utils =
        let ürünOluştur ad fiyat =
            {Ad="iPhone X";Fiyat=fiyat}
        let sepetOluştur ad soyad ürünler =
                {Müşteri={Ad=ad;Soyad=soyad}; Ürünler= ürünler}
```

Tip ve Fonksiyonların Organzasyonu

Modülleri ve alan adlarını tipleri ve fonksiyonları organize etmek için kullanabiliriz. Normalde nesne yönelimli (object oriented) programlama dillerinde tipler bir sınıf olarak tanımlanır ve sınıf tanımı tipin alanları/özellikleri ile birlilkte tipin sağladığı fonksiyonları da içerir. F#'da ise saf fonksiyonel programlama yaparken sınflar kullanılmaz bu nedenle tipleri ve tipler ile ilişkili fonksiyonları organize etmek için iki yöntem kullanılır

Yöntem-1: Tip ve fonksiyonları ayrı ayrı tanımlamak. Bu yöntemde tip tanımı alan adı altında yapılırken, tip ile ilgili fonksiyonlar ise alan adı altında bir iç modül içinde yapılır. Bu yöntem diğer .NET dilleri tarafından kullanılacak olan F# kodlarında tercih edilelidir, çünkü tip isimleri bu yöntemle diğer dil kullanıcıları için açık ve net olarak görünür olur.

```
(* 03_7_02.2.fs *)

// --------- 1. Yöntem ---------
namespace SanalMarket1

// Tip tanımı
type MüşteriTipi = {Ad:string;Soyad:string}

// Tip adını taşıyan modül
module Müşteri =
   // Tip ile ilgili işlem yapan fonksiyon
```

```
   let oluştur ad soyad =
    {Ad=ad;Soyad=soyad}
```

Yöntem-2: Tip ve ilişkili fonksiyonlar beraber tanımlanır. Bu yöntemde modül adı tip adı olarak kullanılır, gerçek tip modül altında basit bir isimle ve fonksiyonlar ile birlikte tanımlanır. Bu yöntem diğer .NET dilleri tarafından kullanılması hedeflenmeyen F# kodlarında tercih edilmelidir. Bu yönetm diğer fonksiyonel dillerdeki iyi uygulama örnekleri (best practice) ile de uyumludur.

```
(* 03_7_02.3.fs *)

// --------- 2. Yöntem ---------
namespace SanalMarket2

// Tipin adını taşıyan modül
module Müşteri =
    // Gerçek tip tanımı basit bir isim ile yapılıyor
    type T = {Ad:string;Soyad:string}

    // Tip ile ilgili işlem yapan fonksiyon
    let oluştur ad soyad =
        {Ad=ad;Soyad=soyad}
```

Hangi yöntem kullanılırsa kullanılsın tip ve ilişkili fonksiyon çağırıları kullanım açısından çok farklı olmaz.

```
(* 03_7_04.fsx *)
#load "03_7_02.2.fs"
#load "03_7_02.3.fs"

// --------- 1. Yöntem TEST---------
open SanalMarket1

let m1 = Müşteri.oluştur "Mahmut" "Tuncer"

printfn "Müşteri %A" m1

// --------- 2. Yöntem TEST---------

open SanalMarket2

let m2 = Müşteri.oluştur "Mahmut" "Tuncer"
printfn "Müşteri %A" m1
```