Fonksiyonlar - Swift

#swift #yazılım

Fonksiyonları tanımlayın ve çağırın, bağımsız değişkenlerini etiketleyin ve dönüş değerlerini kullanın.

Fonksiyonlar, belirli bir görevi gerçekleştiren bağımsız kod parçalarıdır. Bir fonksiyona ne yaptığını tanımlayan bir ad verirsiniz ve bu ad, gerektiğinde görevini yerine getirmesi için fonksiyonu "*çağırmak*" için kullanılır.

Swift'in birleştirilmiş fonksiyon sözdizimi, parametre adları olmayan basit bir C stili fonksiyondan, her parametre için adlar ve bağımsız değişken etiketleri içeren karmaşık Objective-C stili bir yönteme kadar her şeyi ifade edecek kadar esnektir. Parametreler, fonksiyon çağrılarını basitleştirmek için varsayılan değerler sağlayabilir ve fonksiyon yürütmeyi tamamladıktan sonra iletilen bir değişkeni değiştiren giriş-çıkış parametreleri olarak iletilebilir.

Swift'deki her fonksiyonun, fonksiyonun parametre türleri ve dönüş türünden oluşan bir türü vardır. Bu türü Swift'deki diğer türler gibi kullanabilirsiniz, bu da fonksiyonları diğer fonksiyonlara parametre olarak geçirmeyi ve fonksiyonlardan fonksiyon döndürmeyi kolaylaştırır. Fonksiyonlar, kullanışlı işlevselliği iç içe geçmiş bir fonksiyon kapsamı içinde kapsüllemek için diğer fonksiyonların içinde de yazılabilir.

Fonksiyonları Tanımlama ve Çağırma

Bir fonksiyon tanımladığınızda, isteğe bağlı olarak, fonksiyonun girdi olarak aldığı, parametreler olarak bilinen bir veya daha fazla adlandırılmış, yazılı değer tanımlayabilirsiniz . İsteğe bağlı olarak, fonksiyonun bittiğinde çıktı olarak geri ileteceği, dönüş türü olarak bilinen bir değer türü de tanımlayabilirsiniz .

Her fonksiyonun, fonksiyonun gerçekleştirdiği görevi açıklayan bir fonksiyon adı vardır. Bir fonksiyonu kullanmak için, o fonksiyonu adıyla “çağırırsınız” ve fonksiyonun parametre türleriyle eşleşen girdi değerlerini (argümanlar olarak bilinir) iletirsiniz. Bir fonksiyonun bağımsız değişkenleri her zaman fonksiyonun parametre listesiyle aynı sırada sağlanmalıdır.

Aşağıdaki örnekteki fonksiyon olarak adlandırılır `greet(person:)`, çünkü yaptığı budur — girdi olarak bir kişinin adını alır ve o kişi için bir selamlama döndürür. StringBunu başarmak için, bir giriş parametresi (adlandırılan bir değer) ve o kişi için bir selamlama içerecek personbir dönüş türü tanımlarsınız `:String`

Aşağıdaki örnekteki fonksiyona `greet(person:)` denir, çünkü yaptığı şey budur — bir kişinin adını girdi olarak alır ve o kişi için bir selamlama döndürür. Bunu başarmak için,

bir giriş parametresi — `person` adı verilen bir `String` değeri — ve o kişi için bir selamlama içerecek bir `String` dönüş türü tanımlarsınız:

```
func greet(person: String) -> String {
    let greeting = "Hello, " + person + "!"
    return greeting
}
```

Tüm bu bilgiler, `func` anahtar sözcüğünün öneki olan fonksiyonun (*definition*)/*tanımına* toplanır. Fonksiyonun dönüş türünü *dönüş oku* `->` (kısa çizgi ve ardından dik açılı ayraç) ile belirtirsiniz, ardından döndürülecek türün adı gelir.

Tanım, fonksiyonun ne yaptığını, ne almayı beklediğini ve bittiğinde ne döndürdüğünü açıklar. Tanım, fonksiyonun kodunuzun herhangi bir yerinden açık bir şekilde çağrılmasını kolaylaştırır:

```
print(greet(person: "Anna"))
// Prints "Hello, Anna!"
print(greet(person: "Brian"))
// Prints "Hello, Brian!"
```

Fonksiyonu, bağımsız değişken etiketinden sonra greet(person:)bir değer ileterek çağırırsınız , örneğin . Fonksiyon bir değer döndürdüğü için, yukarıda gösterildiği gibi bu dizeyi yazdırmak ve dönüş değerini görmek için fonksiyona yapılan bir çağrıya sarılabilir `.Stringpersongreet(person: "Anna")Stringgreet(person:)`

`Greet(person: "Anna")` gibi `person` bağımsız değişken etiketinden sonra bir `String` değeri geçirerek `great(person:)` fonksiyonunu çağırırsınız. Fonksiyon bir `String` değeri döndürdüğünden, `greet(person:)`, bu dizeyi yazdırmak ve yukarıda gösterildiği gibi dönüş değerini görmek için `print(_:separator:terminator:)` fonksiyonuna yapılan bir çağrıya sarılabilir.

```
print(_:separator:terminator:)
```

Not: `Print(_:separator:terminator:)` fonksiyonunun ilk bağımsız değişkeni için bir etiketi yoktur ve diğer bağımsız değişkenleri varsayılan bir değere sahip oldukları için isteğe bağlıdır. Fonksiyon sözdizimindeki bu varyasyonlar aşağıda Function Argument Labels and Parameter Names ve Default Parameter Values.

Fonksiyonun gövdesi, adında yeni bir sabit `greet(person:)` tanımlayarak ve onu basit bir karşılama mesajı olarak ayarlayarak başlar. Bu selamlama daha sonra anahtar sözcük kullanılarak fonksiyondan geri geçirilir . Yazan kod satırında , fonksiyon yürütmesini bitirir ve geçerli değerini döndürür .`Stringgreetingreturnreturn greetinggreeting`

greet(person:) Fonksiyonu farklı giriş değerleriyle birden çok kez çağırabilirsiniz . "Anna" Yukarıdaki örnek, bir giriş değeri ve bir giriş değeri ile çağrıldığında ne olacağını gösterir "Brian". Fonksiyon, her durumda özel bir selamlama döndürür.

`greet(person:)` fonksiyonunun gövdesi, `greeting` adı verilen yeni bir `String` sabiti tanımlayarak ve onu basit bir selamlama mesajına ayarlayarak başlar. Bu selamlama daha sonra `return` anahtar sözcüğü kullanılarak fonksiyondan geri aktarılır. `return greeting` yazan kod satırında, fonksiyon yürütülmesini tamamlar ve `greeting` geçerli değerini döndürür.

Bu fonksiyonun gövdesini kısaltmak için mesaj oluşturma ve dönüş ifadesini tek bir satırda birleştirebilirsiniz:

```
func greetAgain(person: String) -> String {
        return "Hello again, " + person + "!"
        }
print(greetAgain(person: "Haluk"))
// Prints "Hello again, Haluk!"
```

Fonksiyon Parametreleri ve Dönüş Değerleri

Swift'te fonksiyon parametreleri ve dönüş değerleri son derece esnektir. Tek bir adsız parametreye sahip basit bir yardımcı fonksiyondan, anlamlı parametre adları ve farklı parametre seçeneklerine sahip karmaşık bir fonksiyona kadar her şeyi tanımlayabilirsiniz.

Fonksiyon Parametreleri ve Dönüş Değerleri

Swift'te fonksiyon parametreleri ve dönüş değerleri son derece esnektir. Tek bir isimsiz parametreye sahip basit bir yardımcı program fonksiyonundan, etkileyici parametre adlarına ve farklı parametre seçeneklerine sahip karmaşık bir fonksiyona kadar her şeyi tanımlayabilirsiniz.

Parametresiz Fonksiyonlar

Giriş parametrelerini tanımlamak için fonksiyonlara gerek yoktur. İşte giriş parametresi olmayan, çağrıldığında her zaman aynı `String` mesajını döndüren bir fonksiyon:

```
func sayHelloWorld() -> String {
    return "hello, world"
}
print(sayHelloWorld())
// Prints "hello, world"
```

Fonksiyon tanımı, herhangi bir parametre almasa bile, fonksiyon adından sonra parantezlere ihtiyaç duyar. Fonksiyon çağrıldığında fonksiyon adından sonra boş bir parantez çifti gelir.

Çoklu Parametrelere Sahip Fonksiyonlar

Fonksiyonlar, fonksiyonun parantezleri içinde virgülle ayrılmış şekilde yazılan birden çok giriş parametresine sahip olabilir.

Bu fonksiyon, bir kişinin adını ve daha önce karşılanıp karşılanmadığını girdi olarak alır ve o kişi için uygun bir karşılama döndürür:

```
func greet(person: String, alreadyGreeted: Bool) -> String {
    if alreadyGreeted {
        return greetAgain(person: person)
    } else {
        return greet(person: person)
    }
}
print(greet(person: "Tim", alreadyGreeted: true))
// Prints "Hello again, Tim!"
```

`Great(person:already Greeted:)` fonksiyonunu, hem `person` etiketli bir `String` bağımsız değişken değerini hem de virgülle ayrılmış parantez içinde `alreadyGreeted` etiketli bir `Bool` bağımsız değişken değerini ileterek çağırırsınız. Bu fonksiyonun önceki bir bölümde gösterilen `greet(person:)` fonksiyonundan farklı olduğunu unutmayın. Her iki fonksiyonun da `greet` ile başlayan adları olmasına rağmen, `greet(person:alreadyGreeted:)` fonksiyonu iki argüman alır, ancak `greet(person:)` fonksiyonu yalnızca bir argüman alır.

Dönüş Değerleri Olmayan Fonksiyonlar

Bir dönüş türü tanımlamak için fonksiyonlar gerekli değildir. İşte fonksiyonun , kendi değerini döndürmek yerine `greet(person:)` kendi değerini yazdıran bir sürümü `:String`

```
func greet(person: String) {
    print("Hello, \(person)!")
}
greet(person: "Dave")
// Prints "Hello, Dave!"
```

Bir değer döndürmesi gerekmediğinden, fonksiyonun tanımı dönüş okunu (→) veya dönüş türünü içermez.

Not `greet(person:)` Açıkça söylemek gerekirse, fonksiyonun bu sürümü, dönüş değeri tanımlanmasa bile yine de bir değer döndürür . Tanımlanmış bir dönüş türü olmayan fonksiyonlar, özel bir tür değeri döndürür Void. Bu sadece olarak yazılan boş bir demettir ().

Kesin olarak konuşursak, `greet(person:)` fonksiyonunun bu sürümü, *dönüş değeri tanımlanmamış olsa da* yine de bir değer döndürür. Tanımlanmış bir dönüş türü olmayan fonksiyonlar, `Void` türünde özel bir değer döndürür. Bu sadece `()` olarak yazılan boş bir demettir.

Bir fonksiyonun dönüş değeri, çağrıldığında göz ardı edilebilir:

```
func printAndCount(string: String) -> Int {
    print(string)
    return string.count
}
func printWithoutCounting(string: String) {
    let _ = printAndCount(string: string)
}
printAndCount(string: "hello, world")
// prints "hello, world" and returns a value of 12
printWithoutCounting(string: "hello, world")
// prints "hello, world" but doesn't return a value
```

İlk fonksiyon olan `printAndCount (string:)` bir string yazdırır ve ardından karakter sayısını `Int` olarak döndürür. İkinci fonksiyon, `printWithoutCounting(string:)`, ilk fonksiyonu çağırır, ancak dönüş değerini yok sayar. İkinci fonksiyon çağrıldığında, mesaj yine de ilk fonksiyon tarafından yazdırılır, ancak döndürülen değer kullanılmaz.

Not : Dönüş değerleri yok sayılabilir, ancak bir değer döndüreceğini söyleyen bir fonksiyon her zaman bunu yapmalıdır. Tanımlı bir dönüş türüne sahip bir fonksiyon, bir değer döndürmeden kontrolün fonksiyonun altından düşmesine izin veremez ve bunu yapmaya çalışmak derleme zamanı hatasına neden olur.

Çoklu Dönüş Değerlerine Sahip Fonksiyonlar

Bir fonksiyonun tek bir bileşik dönüş değerinin parçası olarak birden çok değer döndürmesi için dönüş türü olarak bir demet türü kullanabilirsiniz.

Aşağıdaki örnek, `Int` değerleri dizisindeki en küçük ve en büyük sayıları bulan `minMax(array:)` adlı bir fonksiyonu tanımlar:

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

`minMax(array:)` fonksiyonu, iki `Int` değeri içeren bir demet döndürür. Bu değerler, fonksiyonun dönüş değerini sorgularken ada göre erişilebilmeleri için `min` ve `maks` olarak etiketlenir.

`minMax(array:)` fonksiyonunun gövdesi, `currentMin` ve `currentMax` adlı iki çalışma değişkenini dizideki ilk tamsayının değerine ayarlayarak başlar. Fonksiyon daha sonra dizideki kalan değerler üzerinde yinelenir ve sırasıyla `currentMin` ve `currentMax` değerlerinden daha küçük veya daha büyük olup olmadığını görmek için her değeri kontrol eder. Son olarak, toplam minimum ve maksimum değerler iki Int değerinden oluşan bir demet olarak döndürülür.

Grubun üye değerleri, fonksiyonun dönüş türünün bir parçası olarak adlandırıldığından, bulunan minimum ve maksimum değerleri almak için bunlara nokta sözdizimi ile erişilebilir:

```
let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \(bounds.max)")
// Prints "min is -6 and max is 109"
```

Tuple üyelerinin, tuple'ın fonksiyondan döndürüldüğü noktada adlandırılması gerekmediğini unutmayın, çünkü adları zaten fonksiyonun dönüş türünün bir parçası olarak belirtilmiştir.

Optional Tuple Dönüş Türleri

Bir fonksiyondam döndürülecek tuple türü, tüm tuple için “no value” olma potansiyeline sahipse, tüm tuple'ın `nil` olabileceği gerçeğini yansıtmak için bir *optional tupl* dönüş türü kullanabilirsiniz. Tuple türünün kapanış parantezinden sonra `(Int, Int)?` veya `(String, Int, Bool)?` gibi bir soru işareti koyarak isteğe bağlı bir demet dönüş türü yazarsınız.

⚠ NOT :

`(Int, Int)` ? gibi optional types bir tuple türü, `(Int?, Int?)` gibi optional types türler içeren bir tuple'dan farklıdır. Optional tuple type, yalnızca tuple içindeki her bir değer değil, tüm tuple optional'dır.

Yukarıdaki `minMax(array:)` fonksiyonu, iki `Int` değeri içeren bir tuple döndürür. Ancak fonksiyon, iletildiği dizide herhangi bir güvenlik denetimi gerçekleştirmez. `array` bağımsız değişkeni boş bir array içeriyorsa, yukarıda tanımlandığı gibi `minMax(dizi:)` fonksiyonu, `array[0]` erişmeye çalışırken bir çalışma zamanı hatasını tetikler.

Boş bir array safely bir şekilde işlemek için, `minMax(array:)` fonksiyonunu optional bir tuple dönüş türüyle yazın ve tuple boşken `nil` değerini döndürün:

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

`minMax(array:)` fonksiyonunun bu sürümünün gerçek bir tuple değeri mi yoksa `nil` m, döndürdüğünü kontrol etmek için optional binding(bağlayıcı) kullanabilirsiniz:

```
if let bounds = minMax(array: [8, -6, 2, 109, 3, 71]) {
    print("min is \(bounds.min) and max is \(bounds.max)")
}
// Prints "min is -6 and max is 109"
```

Örtük Dönüşlü Fonksiyonlar

Fonksiyonun tüm gövdesi tek bir ifade ise, fonksiyon örtük olarak bu ifadeyi döndürür. Örneğin, aşağıdaki her iki fonksiyon de aynı davranışa sahiptir:

```
func greeting(for person: String) -> String {
    "Hello, " + person + "!"
```

```
}
print(greeting(for: "Dave"))
// Prints "Hello, Dave!"

func anotherGreeting(for person: String) -> String {
    return "Hello, " + person + "!"
}
print(anotherGreeting(for: "Dave"))
// Prints "Hello, Dave!"
```

- `greeting(for:)` fonksiyonunun tüm tanımı, döndürdüğü karşılama mesajıdır, yani bu daha kısa biçimi kullanabilir.
- `anotherGreeting(for:)` fonksiyonu, daha uzun bir fonksiyon gibi `return` anahtar sözcüğünü kullanarak aynı karşılama iletisini döndürür. Yalnızca bir `return` satırı olarak yazdığınız herhangi bir fonksiyon, dönüşü atlayabilir.

Shorthand Getter Beyanında 'da göreceğiniz gibi, mülk alıcıları aynı zamanda örtük bir getiri de kullanabilir.

> ⚠ NOT :
>
> Örtük bir dönüş değeri olarak yazdığınız kodun bir değer döndürmesi gerekir. Örneğin, örtülü bir dönüş değeri olarak `print(13)` kullanamazsınız. Ancak, asla `fatalError("Oh no!")` örtük bir dönüş değeri olarak, çünkü Swift örtük dönüşün gerçekleşmediğini bilir.

Fonksiyon Argüman Etiketleri ve Parametre Adları

Her fonksiyon parametresinin hem bir *argüman etiketi* hem de bir *parametre adı* vardır. Bağımsız değişken etiketi, fonksiyonu çağırırken kullanılır; her argüman, fonksiyon çağrısına, ondan önce bağımsız değişken etiketiyle yazılır. Parametre adı fonksiyonun uygulanmasında kullanılır. Varsayılan olarak, parametreler argüman etiketi olarak parametre adlarını kullanır.

```
func someFunction(first Parameter Name: Int, second Parameter Name: Int) {
    // İşlev gövdesinde, first Parameter Name ve second Parameter Name
    // bfirst and second parameter'lerin bağımsız değişken değerlerine
bakın.
}
some Function(first Parameter Name: 1, second Parameter Name: 2)
```

Tüm parametrelerin benzersiz isimleri olmalıdır. Birden fazla parametrenin aynı argüman etiketine sahip olması mümkün olsa da, benzersiz bağımsız değişken etiketleri kodunuzu daha okunabilir hale getirmeye yardımcı olur.

Bağımsız Değişken Etiketlerini Belirleme

Parametre adından önce bir boşlukla ayrılmış bir bağımsız değişken etiketi yazarsınız:

```
func someFunction(argument Label parameterName: Int) {
// Fonksiyon gövdesinde parameterName, o parametrenin bağımsız değişken değerini ifade eder.
}
```

İşte bir kişinin adını ve memleketini alan ve bir selamlama döndüren `greet(person:)` fonksiyonunun bir varyasyonu:

```
func greet(person: String, from hometown: String) -> String {
    return "Hello \(person)!  Glad you could visit from \(hometown)."
}
print(great(person: "Bill", from: "Cupertino"))
// Prints "Hello Bill!  Glad you could visit from Cupertino."
```

Argüman etiketlerinin kullanılması, bir fonksiyonun etkileyici, cümle benzeri bir şekilde çağrılmasına izin verirken, yine de amaç olarak okunabilir ve net bir fonksiyon gövdesi sağlayabilir.

Bağımsız Değişken Etiketlerini Atlamak

Bir parametre için bağımsız değişken etiketi istemiyorsanız, bu parametre için açık bir bağımsız değişken etiketi yerine bir alt çizgi/underscore (_) yazın.

```
func someFunction(_ first Parameter Name: Int, second Parameter Name: Int)
{
        // İşlev gövdesinde, firstParameterName ve secondParameterName,
first ve second parametrelerin bağımsız değişken değerlerine başvurur.
}
some Function(1, second Parameter Name: 2)
```

Bir parametrenin bağımsız değişken etiketi varsa, fonksiyonu çağırdığınızda argümanın etiketlenmesi *gerekir*.

Varsayılan Parametre Değerleri

Bu parametrenin türünden sonra parametreye bir değer atayarak bir fonksiyondaki herhangi bir parametre için *varsayılan* bir *değer* tanımlayabilirsiniz. Varsayılan bir değer tanımlanırsa, fonksiyonu çağırırken bu parametreyi atlayabilirsiniz.

```
func someFunction(parameter Without Default: Int, parameter With Default:
Int = 12) {
        // Bu fonksiyonu çağırırken ikinci bağımsız değişkeni atlarsanız,
fonksiyon gövdesi içindeki parameterWithDefault değeri 12'dir.
}
some Function(parameter Without Default: 3, parameter With Default: 6) //
varsayılan parametre 6'dır
some Function(parameter Without Default: 4) // varsayılan parametre 12'dir
```

Varsayılan değerleri olmayan parametreleri, bir fonksiyonun parametre listesinin başına, varsayılan değerleri olan parametrelerin önüne yerleştirin. Varsayılan değerlere sahip olmayan parametreler genellikle fonksiyonun anlamı için daha önemlidir — önce bunları yazmak, herhangi bir varsayılan parametrenin atlanıp atlanmadığına bakılmaksızın aynı fonksiyonun çağrıldığını tanımayı kolaylaştırır.

Değişken Parametreler

Değişken bir *parametre,* belirli bir türde sıfır veya daha fazla değeri kabul eder. Fonksiyon çağrıldığında parametrenin değişen sayıda giriş değeri iletilebileceğini belirtmek için değişken bir parametre kullanırsınız. Parametrenin tür adından sonra üç nokta karakteri `(...)` ekleyerek değişken parametreler yazın.

Değişken bir parametreye geçirilen değerler, fonksiyonun gövdesinde uygun türde bir array olarak kullanılabilir hale getirilir. Örneğin, `numbers` ve type'ı `Double...` olan değişken bir parametre, fonksiyonun gövdesi içinde `[Double]` türünde `numbers` adı verilen sabit/constant bir array olarak kullanılabilir hale getirilir.

Aşağıdaki örnek, herhangi bir uzunluktaki sayıların bir listesi için *aritmetik ortalamayı* (*ortalama* olarak da bilinir) hesaplar:

Bu Swift fonksiyonu, değişen sayıda bağımsız değişkenleri olan bir ortalama hesaplama fonksiyonu gerçekleştirir. Fonksiyonun imzası şu şekildedir:

```
func arithmetic Mean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
```

```
}
arithmetic Mean(1, 2, 3, 4, 5)
// bu beş sayının aritmetik ortalaması olan 3.0'ı döndürür
arithmetic Mean(3, 8.25, 18.75)
// bu üç sayının aritmetik ortalaması olan 10.0'ı döndürür
```

Burada, `_ numbers` ifadesi, fonksiyonun alacağı değişen sayıda (0'dan fazla) `Double` türünden bağımsız değişkenleri temsil eder.

Fonksiyon, aldığı tüm sayıları toplamak için bir döngü kullanır ve toplamı `total` adlı bir değişkende saklar. Daha sonra, `numbers` dizisinin eleman sayısına bölerek sayıların aritmetik ortalamasını hesaplar ve bu değeri geri döndürür.

`arithmeticMean(1, 2, 3, 4, 5)

Fonksiyona 5 sayıyı argüman olarak geçirir ve sonuç olarak 3.0 (1+2+3+4+5=15, 15/5=3) değerini döndürür.

Benzer şekilde, şu kod örneği:

`arithmeticMean(3, 8.25, 18.75)`

Bir fonksiyon birden fazla değişken parametreye sahip olabilir. Değişken bir parametreden sonra gelen ilk parametrenin bir argüman etiketi olmalıdır. Argüman etiketi, hangi argümanların değişken parametreye geçirildiğini ve hangi argümanların değişken parametreden sonra gelen parametrelere geçirildiğini açık hale getirir.

Giriş-Çıkış Parametreleri

Swift dilinde in-out parameters (içeri-dışarı parametreleri), bir fonksiyona argüman olarak geçirilen değerlerin fonksiyon içinde değiştirilip dışarıda kullanılabilmesini sağlar.

Bir fonksiyon, içeri-dışarı parametrelerle çağrıldığında, parametreler `in-out` olarak belirtilir. Bu tür bir parametre, `"&"` işaretiyle öne eklenerek tanımlanır. Örneğin:

```
func swapInts(_ a: inout Int, _ b: inout Int) {
    let temp = a
    a = b
    b = temp
}
```

Bu fonksiyon, "a" ve "b" adlı iki tamsayı içeri-dışarı parametreleri alır ve bunları birbirleriyle değiştirir. Fonksiyon, "temp" adlı geçici bir değişken kullanarak "a" ve "b" değerlerini değiştirir.

Bu fonksiyonu kullanmak için, bir fonksiyon çağrısı yapılırken, "&" işaretiyle öne eklenerek "a" ve "b" parametreleri işaret edilir:

```
var x = 10
var y = 20
swapInts(&x, &y)
print("x: \(x), y: \(y)") // prints "x: 20, y: 10"
```

Bu örnekte, `x` ve y adlı iki değişken, "swapInts" fonksiyonuna argüman olarak geçilir. Fonksiyon, `x` ve `y` değişkenlerinin değerlerini değiştirir, bu nedenle sonuç olarak `x` 20 ve `y` 10 olarak yazdırılır.

İçeri-dışarı parametreleri kullanmak, özellikle büyük veri yapıları veya nesnelerle çalışırken, daha verimli bir programlama yöntemi olabilir. Bununla birlikte, içeri-dışarı parametrelerin kullanımı, fonksiyonların doğrudan etkilemesi gereken veriler üzerinde kontrollü bir şekilde çalışmak için biraz daha fazla dikkat gerektirebilir.

Fonksiyon parametreleri varsayılan olarak sabittir. Bir fonksiyon parametresinin değerini o fonksiyonun gövdesinden değiştirmeye çalışmak, bir derleme zamanı hatasına neden olur. Bu, bir parametrenin değerini yanlışlıkla değiştiremeyeceğiniz anlamına gelir. Bir fonksiyonun, bir parametrenin değerini değiştirmesini istiyorsanız ve fonksiyon çağrısı sona erdikten sonra bu değişikliklerin devam etmesini istiyorsanız, bunun yerine bu parametreyi bir *in-out parametresi* olarak tanımlayın.

`Inout` anahtar sözcüğünü bir parametrenin türünden hemen önce yerleştirerek bir in-out parametresi yazarsınız. Bir ın-out parametresi, fonksiyona *in* geçirilen, fonksiyon tarafından değiştirilen ve orijinal değeri değiştirmek için fonksiyondan *out* geri geçirilen bir değere sahiptir. Çıkış parametrelerinin davranışı ve ilişkili derleyici optimizasyonları hakkında ayrıntılı bir tartışma için bkz. Çıkış Parameters.

Bir değişkeni yalnızca bir in-out parametresi için argüman olarak geçirebilirsiniz. Argüman olarak bir sabit veya değişmez bir değer geçiremezsiniz, çünkü sabitler ve değişmezler değiştirilemez. Fonksiyon tarafından değiştirilebileceğini belirtmek için bir in-out parametresine argüman olarak ilettiğinizde, bir değişkenin adının hemen önüne bir ampersand (`&`) yerleştirirsiniz.

> ⚠ **NOT :**
>
> In-out parametreleri varsayılan değerlere sahip olamaz ve değişken parametreler `inout` olarak işaretlenemez.

İşte `a` ve `b` olarak adlandırılan iki in-out tamsayı parametresine sahip `swapTwoInts(_:_:)` adlı bir fonksiyona bir örnek:

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporary
}
```

`swapTwoInts(_:_:)` fonksiyonu, `b`'nin değerini `a`'ya ve `a`'nın değerini `b`'ye değiştirir. Fonksiyon, `a`'nın değerini geçici olarak adlandırılan geçici bir sabitte depolayarak, `b`'nin değerini `a`'ya atayarak ve ardından geçici değeri `b`'ye atayarak bu takası gerçekleştirir.

Değerlerini değiştirmek için `Int` türünde iki değişkenle `swapTwoInts(_:_:)` fonksiyonunu çağırabilirsiniz. `swapTwoInts(_:_:)` fonksiyonuna iletildiklerinde `someInt` ve `anotherInt` adlarının bir ve işareti ile öneklendiğini unutmayın:

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// Prints "someInt şimdi 107 ve anotherInt şimdi 3"
```

Yukarıdaki örnek, `someInt` ve `anotherInt` orijinal değerlerinin, orijinal olarak fonksiyonun dışında tanımlanmış olsalar bile `swapTwoInts(_:_:)` fonksiyonu tarafından değiştirildiğini göstermektedir.

> ⚠ NOT :
>
> In-ou parametreleri, bir fonksiyondan bir değer döndürmekle aynı şey değildir. Yukarıdaki `swapTwoInts` örneği bir dönüş türü tanımlamaz veya bir değer döndürmez, ancak yine de `someInt` ve `anotherInt' değerlerini değiştirir. In-out parametreleri, bir fonksiyonun fonksiyon gövdesinin kapsamı dışında bir etkiye sahip olmasının alternatif bir yoludur.

İşlev Türleri

Her fonksiyonun, parametre türlerinden ve fonksiyonun dönüş türünden oluşan belirli bir *fonksiyon türü* vardır.

Örneğin:

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}
```

```
func multiplyTwoInt(_ a: Int, _ b: Int) -> Int {
    return a * b
}
print(addTwoInts(2, 3))       // 5
print(multiplyTwoInt(2, 3))   // 6
```

Bu örnek, `addTwoInts` ve `multiplyTwoInts` adı verilen iki basit matematiksel fonksiyonu tanımlar. Bu fonksiyonların her biri iki `Int` değeri alır ve uygun bir matematiksel işlemin gerçekleştirilmesinin sonucu olan bir `Int` değeri döndürür.

Bu fonksiyonların her ikisinin de türü `(Int, Int) -> Int`. Bu şu şekilde okunabilir:

> Her ikisi de `Int` türünde iki parametreye sahip olan ve `Int` türünde bir değer döndüren bir fonksiyon.

`addTwoInts(_:_:)` fonksiyonu, `mathFunction` değişkeniyle aynı türe sahiptir ve bu nedenle bu atamaya Swift'in tür denetleyicisi tarafından izin verilir.

Artık atanan fonksiyonu `mathFunction` adıyla çağırabilirsiniz`:

```
print("Result: \(mathFunction(2, 3))")
// Prints "Result: 5"
```

Aynı eşleşen türe sahip farklı bir fonksiyon, aynı değişkene, fonksiyonsuz türlerde olduğu gibi atanabilir:

```
mathFunction = multiplyTwoInts
print("Result: \(mathFunction(2, 3))")
// Prints "Result: 6"
```

Diğer herhangi bir türde olduğu gibi, bir sabite veya değişkene bir fonksiyon atadığınızda fonksiyon türünü çıkarmak için Swift'e bırakabilirsiniz:

```
let anotherMathFunction = addTwoInts
// anotherMathFunction türünde olduğu sonucuna varılır (Int, Int) -> Int
```

Parametre Türleri Olarak Fonksiyon Türleri

`(Int, Int) -> Int` gibi bir fonksiyon tipini başka bir fonksiyon için parametre tipi olarak kullanabilirsiniz. Bu, bir fonksiyonun uygulamasının bazı yönlerini, fonksiyon çağrıldığında sağlaması için fonksiyon arayan kişiye bırakmanıza olanak tanır.

İşte matematik fonksiyonlarının sonuçlarını yukarıdan yazdırmak için bir örnek

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
        return a + b
}

func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b:
Int) {
        print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
```

Bu örnek, üç parametresi olan `printMathResult(_:_:_:)` adlı bir fonksiyonu tanımlar. İlk parametre `mathFunction` olarak adlandırılır ve `(Int, Int) -> Int` türündedir. Bu türdeki herhangi bir fonksiyonu bu ilk parametre için bağımsız değişken olarak iletebilirsiniz. İkinci ve üçüncü parametreler `a` ve `b` olarak adlandırılır ve her ikisi de `Int` türündedir. Bunlar, sağlanan matematik fonksiyonu için iki giriş değeri olarak kullanılır.

`printMathResult(_:_:_:)` çağrıldığında, `addTwoInts(_:_:)` fonksiyonu ve `3` ve `5` tamsayı değerleri iletilir. Sağlanan fonksiyonu `3` ve `5` değerleriyle çağırır ve `8` sonucunu yazdırır.

> `printMathResult(_:_:_:)` fonksiyonu, uygun türde bir matematik fonksiyonuna yapılan çağrının sonucunu yazdırmaktır. Bu fonksiyonun uygulamasının gerçekte ne yaptığı önemli değil - yalnızca fonksiyonun doğru türde olması önemlidir. Bu, `printMathResult(_:_:_:)` fonksiyonunun bir kısmını fonksiyonun çağıranına tür açısından güvenli bir şekilde devretmesini sağlar.

Dönüş Türleri Olarak Fonksiyon Türleri

Başka bir fonksiyonun dönüş türü olarak bir fonksiyon türü kullanabilirsiniz. Bunu, dönen fonksiyonun dönüş okundan (`->`) hemen sonra tam bir fonksiyon türü yazarak yaparsınız.

Bir sonraki örnek, `stepForward(_:)` ve `stepBackward(_:)` olarak adlandırılan iki basit fonksiyonu tanımlar. `stepForward(_:)` fonksiyonu, input değerinden bir değer daha döndürür ve `stepBackward(_:)` fonksiyonu, input değerinden bir değer daha az döndürür. Her iki fonksiyonun da bir türü vardır `(Int) -> Int`:

```
func stepForward(_ input: Int) -> Int {
    return input + 1
```

```
}
func stepBackward(_ input: Int) -> Int {
    return input - 1
}
```

İşte dönüş türü `(Int) -> Int` olan `chooseStepFunction(backward:)` adlı bir fonksiyon. `chooseStepFunction(backward:)` fonksiyonu, `stepForward(_:)` adlı bir Boole parametresine dayalı olarak `stepForward(_:)` fonksiyonunu veya `stepBackward(_:)` fonksiyonunu döndürür:

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
        return backward ? stepBackward : stepForwar
}
```

Artık bir yöne veya diğerine adım atacak bir fonksiyon elde etmek için `chooseStepFunction(backward:)` fonksiyonunu kullanabilirsiniz:

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
// moveNearerToZero now refers to the stepBackward() function
```

Yukarıdaki örnek, `currentValue` adlı bir değişkeni aşamalı olarak sıfıra yaklaştırmak için pozitif veya negatif bir adımın gerekli olup olmadığını belirler. `currentValue` `3` başlangıç değerine sahiptir, yani `currentValue> 0` `true` değerini döndürür ve `chooseStepFunction(backward:)` fonksiyonunun `stepBackward(_:)` fonksiyonunu döndürmesine neden olur. Döndürülen fonksiyona yapılan bir başvuru, `moveNearerToZero` adlı bir sabitte saklanır.

Artık `moveNearerToZero` doğru fonksiyona atıfta bulunduğuna göre, sıfıra saymak için kullanılabilir:

```
print("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// 3...
// 2...
// 1...
// zero!
```

İç İçe Fonksiyonlar

Bu bölümde şimdiye kadar karşılaştığınız tüm fonkasiyonlar, küresel kapsamda tanımlanan *genel fonksiyonlara* örnek olmuştur. *İç içe fonksiyonlar* olarak bilinen diğer fonksiyonların gövdelerinin içindeki fonksiyonları da tanımlayabilirsiniz.

İç içe geçmiş fonksiyonlar varsayılan olarak dış dünyadan gizlenir, ancak yine de çevreleyen fonksiyonları tarafından çağrılabilir ve kullanılabilir. Bir çevreleyen fonksiyon, iç içe geçmiş fonksiyonun başka bir kapsamda kullanılmasına izin vermek için iç içe geçmiş fonksiyonlardan birini de döndürebilir.

İç içe geçmiş fonksiyonları kullanmak ve döndürmek için yukarıdaki `chooseStepFunction(backward:)` örneğini yeniden yazabilirsiniz:

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backward ? stepBackward : stepForward
}
var currentValue = -4
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
// // moveNearerToZero şimdi iç içe geçmiş stepForward()) fonksiyonunu
ifade ediyor
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// -4...
// -3...
// -2...
// -1...
// zero!
```