Temel Bilgiler — Swift

#yazılım #swift

Swift.org

Swift Programlama Dili

Swift 5.6

- Swift'e hoş geldiniz
- Dil Rehberi
  - Temeller
  - Temel Operatörler
  - Dizeler ve Karakterler
  - Koleksiyon Türleri
  - Kontrol akışı
  - Fonksiyonlar
  - Kapanışlar
  - numaralandırmalar
  - Yapılar ve Sınıflar
  - Özellikleri
  - yöntemler
  - Abonelikler
  - Miras
  - başlatma
  - Başlatma
  - Opsiyonel Zincirleme
  - Hata yönetimi
  - eşzamanlılık
  - Tip Döküm
  - İç İçe Türler
  - Uzantılar
  - protokoller
  - jenerik
  - Opak Tipler
  - Otomatik Referans Sayımı
  - Bellek Güvenliği
  - Giriş kontrolu

Temel Bilgiler

Swift, iOS, macOS, watchOS ve tvOS uygulama geliştirme için yeni bir programlama dilidir. Bununla birlikte, Swift'in birçok bölümü, C ve Objective-C'de geliştirme deneyiminizden aşina olacaktır.

Swift, dahil olmak üzere tüm temel C ve Objective-C türlerinin kendi sürümlerini sağlar. `Int` tamsayılar için `Double` ve `Float` kayan noktalı değerler için, `Bool` Boole değerleri için ve `String` metinsel veriler için. Swift ayrıca üç ana koleksiyon türünün güçlü sürümlerini sağlar. `Array`, `Set`, ve `Dictionary` bölümünde açıklandığı gibi Koleksiyon Türleri .

C gibi, Swift de değerleri tanımlayıcı bir adla saklamak ve bunlara atıfta bulunmak için değişkenler kullanır. Swift, değerleri değiştirilemeyen değişkenleri de kapsamlı bir şekilde kullanır. Bunlar sabitler olarak bilinir ve C'deki sabitlerden çok daha güçlüdür. Değişmesi gerekmeyen değerlerle çalışırken kodu daha güvenli ve daha net hale getirmek için Swift genelinde sabitler kullanılır.

Bilinen türlere ek olarak Swift, tanımlama grupları gibi Objective-C'de bulunmayan gelişmiş türleri de sunar. Tuple'lar, değer gruplamaları oluşturmanıza ve çevresinden geçmenize olanak tanır. Bir işlevden birden çok değeri tek bir bileşik değer olarak döndürmek için bir tanımlama grubu kullanabilirsiniz.

Swift ayrıca bir değerin yokluğunu işleyen isteğe bağlı türleri de sunar. Seçenekler ya "bir değer var *ve* eşittir *x'e* " ya da " *hiç* bir değer yok" der. Opsiyonelleri kullanmak, kullanmaya benzer `nil` Objective-C'deki işaretçilerle, ancak yalnızca sınıflar için değil, herhangi bir tür için çalışırlar. Seçenekler yalnızca daha güvenli ve daha etkileyici olmakla kalmaz `nil` Objective-C'deki işaretçiler, Swift'in en güçlü özelliklerinin çoğunun kalbinde yer alır.

Swift, *tür açısından güvenli* dildir; bu, dilin, kodunuzun çalışabileceği değer türleri konusunda net olmanıza yardımcı olduğu anlamına gelir. Kodunuzun bir kısmı bir `String`, tip güvenliği, onu geçmenizi engeller ve `Int` yanlışlıkla. Aynı şekilde, tür güvenliği de yanlışlıkla isteğe bağlı bir `String` isteğe bağlı olmayan bir kod parçasına `String`. Tür güvenliği, geliştirme sürecinde hataları mümkün olduğunca erken yakalamanıza ve düzeltmenize yardımcı olur.

Sabitler ve Değişkenler

Sabitler ve değişkenler bir adı ilişkilendirir (örneğin `maximumNumberOfLoginAttempts` veya `welcomeMessage`) belirli bir türe ait bir değerle

(sayı gibi `10` veya dize `"Hello"`). Bir *sabitin* kez ayarlandıktan sonra değiştirilemezken, bir *değişken* gelecekte farklı bir değere ayarlanabilir.

Sabitleri ve Değişkenleri Bildirmek

Sabitler ve değişkenler kullanılmadan önce bildirilmelidir. ile sabitleri bildirirsiniz `let` anahtar kelime ve değişkenler ile `var` anahtar kelime. Bir kullanıcının yaptığı oturum açma denemelerinin sayısını izlemek için sabitlerin ve değişkenlerin nasıl kullanılabileceğine dair bir örnek:

```
let maximumNumberOfLoginAttempts = 10
var currentLoginAttempt = 0
```

Bu kod şu şekilde okunabilir:

"adlı yeni bir sabit bildir `maximumNumberOfLoginAttempts` ve ona bir değer verin `10`. Ardından, adlı yeni bir değişken tanımlayın `currentLoginAttempt` ve ona bir başlangıç değeri verin `0`."

Bu örnekte, maksimum değer hiçbir zaman değişmediğinden, izin verilen maksimum oturum açma denemesi sayısı sabit olarak bildirilir. Geçerli oturum açma girişimi sayacı bir değişken olarak bildirilir, çünkü bu değer her başarısız oturum açma girişiminden sonra artırılmalıdır.

Tek bir satırda virgülle ayırarak birden çok sabit veya birden çok değişken bildirebilirsiniz:

```
var x = 0.0, y = 0.0, z = 0.0
```

Not: Kodunuzda saklanan bir değer değişmezse, onu her zaman ile sabit olarak bildirin. `let` anahtar kelime. Değişkenleri yalnızca değişmesi gereken değerleri depolamak için kullanın.

Tip Açıklamaları

için bir *tür ek açıklaması* Bir sabit veya değişken bildirirken, sabitin veya değişkenin depolayabileceği değerlerin türü hakkında net olmak Sabit veya değişken adından sonra iki nokta üst üste işareti, ardından bir boşluk ve ardından kullanılacak türün adını koyarak bir tür ek açıklaması yazın.

Bu örnek, adlı bir değişken için bir tür ek açıklaması sağlar. `welcomeMessage`, değişkenin depolayabileceğini belirtmek için `String` değerler:

```
var welcomeMessage: String
```

Beyannamedeki iki nokta üst üste işareti “... türü...” anlamına gelir, bu nedenle yukarıdaki kod şu şekilde okunabilir:

"adlı bir değişken bildirin `welcomeMessage` bu tür `String`.”

"tipi" ifadesi `String` "herhangi bir şeyi saklayabilir" anlamına gelir `String` değer." Bunu, depolanabilecek “tür” (veya “tür”) olarak düşünün.

bu `welcomeMessage` değişken artık hatasız herhangi bir dize değerine ayarlanabilir:

```
welcomeMessage = "Hello"
```

Son değişken adından sonra tek bir tür açıklama ile, virgülle ayrılmış tek bir satırda aynı türden birden çok ilişkili değişken tanımlayabilirsiniz:

```
var red, green, blue: Double
```

Not: Pratikte tür ek açıklamaları yazmanız çok nadirdir. Bir sabit veya değişken için tanımlandığı noktada bir başlangıç değeri sağlarsanız, Swift, Type Safety ve Type Inference . İçinde `welcomeMessage` yukarıdaki örnekte, başlangıç değeri sağlanmamıştır ve bu nedenle `welcomeMessage` değişken, bir başlangıç değerinden çıkarılmak yerine bir tür ek açıklamasıyla belirtilir.

Sabitleri ve Değişkenleri Adlandırma

Sabit ve değişken adları, Unicode karakterler de dahil olmak üzere hemen hemen her karakteri içerebilir:

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"
```

Sabit ve değişken adları boşluk karakterleri, matematiksel semboller, oklar, özel kullanım Unicode skaler değerleri veya çizgi ve kutu çizim karakterleri içeremez. Rakamlar ismin başka yerlerinde yer alabilse de, bir sayı ile de başlayamazlar.

Belirli bir türde bir sabit veya değişken tanımladıktan sonra, onu aynı adla yeniden bildiremez veya farklı türdeki değerleri depolamak için değiştiremezsiniz. Ayrıca bir sabiti bir değişkene veya bir değişkeni bir sabite dönüştüremezsiniz.

Not: Ayrılmış bir Swift anahtar kelimesiyle aynı adı taşıyan bir sabit veya değişken vermeniz gerekiyorsa, anahtar kelimeyi ters tiklerle çevreleyin (`) isim olarak kullanıldığında. Ancak, kesinlikle başka seçeneğiniz olmadığı sürece, anahtar kelimeleri ad olarak kullanmaktan kaçının.

Varolan bir değişkenin değerini, uyumlu türden başka bir değerle değiştirebilirsiniz. Bu örnekte, değeri `friendlyWelcome` değiştirildi `"Hello!"` ile `"Bonjour!"`:

```
var friendlyWelcome = "Hello!"
friendlyWelcome = "Bonjour!"
// friendlyWelcome artık "Bonjour!" oldu.
```

Bir değişkenin aksine, bir sabitin değeri ayarlandıktan sonra değiştirilemez. Bunu yapmaya çalışmak, kodunuz derlendiğinde bir hata olarak bildirilir:

```
let languageName = "Swift"
languageName = "Swift++"
// Bu bir derleme zamanı hatasıdır: languageName değiştirilemez.
```

Sabitleri ve Değişkenleri Yazdırma

Bir sabitin ya da değişkenin geçerli değerini şu komutla yazdırabilirsiniz: `print(_:separator:terminator:)` işlev:

```
print(friendlyWelcome)
// Prints "Bonjour!"
```

bu `print(_:separator:terminator:)` işlev, bir veya daha fazla değeri uygun bir çıktıya yazdıran genel bir işlevdir. Örneğin Xcode'da, `print(_:separator:terminator:)` işlevi, çıktısını Xcode'un “konsol” bölmesinde yazdırır. bu `separator` ve `terminator` parametrenin varsayılan değerleri vardır, bu nedenle bu işlevi çağırdığınızda bunları atlayabilirsiniz. Varsayılan olarak işlev, yazdırdığı satırı bir satır sonu ekleyerek sonlandırır. Satır sonu olmadan bir değer yazdırmak için, sonlandırıcı olarak boş bir dize iletin; örneğin, `print(someValue, terminator: "")`. Varsayılan değerlere sahip parametreler hakkında bilgi için bkz Varsayılan Parametre Değerleri .

Swift *dize enterpolasyonunu* , bir sabitin veya değişkenin adını daha uzun bir dizede yer tutucu olarak dahil etmek ve Swift'den bu sabitin veya değişkenin geçerli değeriyle değiştirmesini istemek Adı parantez içine alın ve açılış parantezinden önce ters eğik çizgi ile yazın:

```
print("The current value of friendlyWelcome is (friendlyWelcome)")
// "The current value of friendlyWelcome is Bonjour!" yazdırır.
```

Not: Dize enterpolasyonu ile kullanabileceğiniz tüm seçenekler Dize .

Comments

Use comments to include nonexecutable text in your code, as a note or reminder to yourself. Comments are ignored by the Swift compiler when your code is compiled.

Swift'deki yorumlar, C'deki yorumlara çok benzer. Tek satırlı yorumlar iki eğik çizgiyle başlar (`//`):

```
// Bu bir yorumdur.
```

Çok satırlı yorumlar eğik çizgiyle başlar ve ardından bir yıldız işareti (`/*`) ve bir yıldız işareti ve ardından bir eğik çizgi ile bitirin (`*/`):

```
\* Bu aynı zamanda bir yorumdur
ancak birden fazla satır üzerine yazılır. */
```

C'deki çok satırlı yorumların aksine, Swift'deki çok satırlı yorumlar diğer çok satırlı yorumların içine yerleştirilebilir. Çok satırlı bir yorum bloğu başlatarak ve ardından ilk blok içinde ikinci bir çok satırlı yorum başlatarak iç içe yorumlar yazarsınız. İkinci blok daha sonra kapatılır, ardından ilk blok gelir:

```
\* Bu, ilk çok satırlı açıklamanın başlangıcıdır.
\* Bu, ikinci iç içe geçmiş çok satırlı yorumdur. */
Bu, ilk çok satırlı yorumun sonu. */
```

İç içe çok satırlı yorumlar, kod zaten çok satırlı yorumlar içeriyor olsa bile, büyük kod bloklarını hızlı ve kolay bir şekilde yorumlamanıza olanak tanır.

Semicolons

Diğer birçok dilden farklı olarak Swift, noktalı virgül yazmanızı gerektirmez (`;`) kodunuzdaki her ifadeden sonra, isterseniz bunu yapabilirsiniz. Ancak, *:* tek bir satıra birden çok ayrı ifade yazmak istiyorsanız noktalı virgül gerekir

let cat = "🐱 "; print(cat)
// Prints "🐱 "

Integers

Tamsayılar , kesirli bileşeni olmayan tam sayılardır, örneğin `42`ve `-23`. Tamsayılar *işaretli* (pozitif, sıfır veya negatif) veya *işaretsizdir* (pozitif veya sıfır).

Swift, 8, 16, 32 ve 64 bit formlarında işaretli ve işaretsiz tamsayılar sağlar. Bu tamsayılar, C'ye benzer bir adlandırma kuralına uyarlar, çünkü 8 bitlik işaretsiz bir tamsayı şu tiptedir: `UInt8`, ve 32 bitlik işaretli bir tamsayı türünde `Int32`. Swift'deki tüm türler gibi, bu tamsayı türlerinin adları büyük harfle yazılır.

Tam Sayı Sınırları

Her tamsayı türünün minimum ve maksimum değerlerine, `min`ve `max`özellikleri:

```
let minValue = UInt8.min // minValue is equal to 0, and is of type UInt8
let maxValue = UInt8.max // maxValue is equal to 255, and is of type UInt8
```

Bu özelliklerin değerleri uygun boyutlu sayı türündedir (örneğin, `UInt8`yukarıdaki örnekte) ve bu nedenle aynı türdeki diğer değerlerle birlikte ifadelerde kullanılabilir.

Int

Çoğu durumda, kodunuzda kullanmak için belirli bir tamsayı boyutu seçmenize gerek yoktur. Swift, ek bir tamsayı türü sağlar, `Int`geçerli platformun yerel sözcük boyutuyla aynı boyuta sahip olan :

- 32 bit platformda, `Int` ile aynı boyutta `Int32`.
- On a 64-bit platform, `Int` is the same size as `Int64`.

Belirli bir tamsayı boyutuyla çalışmanız gerekmedikçe, her zaman `Int` for integer values in your code. This aids code consistency and interoperability. Even on 32-bit platforms, `Int` can store any value between `-2,147,483,648` ve `2,147,483,647`, and is large enough for many integer ranges.

UInt

Swift ayrıca işaretsiz bir tamsayı türü sağlar, `UInt`geçerli platformun yerel sözcük boyutuyla aynı boyuta sahip olan :

- 32 bit platformda, `UInt` ile aynı boyutta `UInt32`.
- 64 bit platformda, `UInt` ile aynı boyutta `UInt64`.

Not: Kullanmak `UInt`yalnızca platformun yerel sözcük boyutuyla aynı boyutta işaretsiz bir tamsayı türüne özellikle ihtiyacınız olduğunda. Eğer durum bu değilse, `Int`saklanacak değerlerin negatif olmadığı bilinse bile tercih edilir. tutarlı bir kullanım `Int`tamsayı değerleri için kodun birlikte çalışabilirliğine yardımcı olur, farklı sayı türleri arasında dönüştürme ihtiyacını ortadan kaldırır ve Tür Güvenliği ve Tür Çıkarımı .

Kayan Noktalı Sayılar

Kayan noktalı sayılar , kesirli bileşene sahip sayılardır, örneğin `3.14159`, `0.1`, ve `-273.15`.

Kayan nokta türleri, tamsayı türlerinden çok daha geniş bir değer aralığını temsil edebilir ve bir veri tabanında saklanabilecekten çok daha büyük veya daha küçük sayıları depolayabilir. `Int`. Swift, iki işaretli kayan noktalı sayı türü sağlar:

- `Double` 64 bitlik kayan noktalı bir sayıyı temsil eder.
- `Float` 32 bitlik kayan noktalı bir sayıyı temsil eder.

Not: `Double` en az 15 ondalık basamak hassasiyetine sahipken, hassasiyeti `Float` 6 ondalık basamak kadar küçük olabilir. Kullanılacak uygun kayan nokta türü, kodunuzda çalışmanız gereken değerlerin doğasına ve aralığına bağlıdır. Her iki türün de uygun olacağı durumlarda, `Double` tercih edilir.

Tip Güvenliği ve Tip Çıkarımı

Swift, *tür açısından güvenli* dildir. Güvenli bir dil, kodunuzun çalışabileceği değer türleri konusunda net olmanızı teşvik eder. Kodunuzun bir kısmı bir `String`, onu geçemezsin `Int`yanlışlıkla.

Swift tip güvenli *tip kontrolleri* olduğundan, kodunuzu derlerken Bu, geliştirme sürecinde hataları mümkün olduğunca erken yakalamanızı ve düzeltmenizi sağlar.

Yazım denetimi, farklı değer türleri ile çalışırken hatalardan kaçınmanıza yardımcı olur. Ancak bu, bildirdiğiniz her sabitin ve değişkenin türünü belirtmeniz gerektiği anlamına gelmez. İhtiyacınız olan değer türünü belirtmezseniz, Swift uygun türü bulmak için *tür çıkarımını* kullanır. Tür çıkarımı, bir derleyicinin, yalnızca sağladığınız değerleri inceleyerek kodunuzu derlerken belirli bir ifadenin türünü otomatik olarak çıkarmasına olanak tanır.

Tür çıkarımı nedeniyle Swift, C veya Objective-C gibi dillerden çok daha az tür bildirimi gerektirir. Sabitler ve değişkenler hala açıkça yazılır, ancak türlerini belirleme işinin çoğu sizin için yapılır.

Tür çıkarımı, özellikle başlangıç değeri olan bir sabit veya değişken bildirdiğinizde kullanışlıdır. atayarak yapılır *değişmez değer* (veya *değişmez* , onu bildirdiğiniz noktada sabite veya değişkene (Sabit değer, doğrudan kaynak kodunuzda görünen bir değerdir, örneğin: `42`ve `3.14159`Aşağıdaki örneklerde.)

Örneğin, değişmez bir değer atarsanız `42`ne tür olduğunu söylemeden yeni bir sabite, Swift, sabitin bir `Int`, çünkü onu bir tamsayıya benzeyen bir sayı ile başlattınız:

```
let meaningOfLife = 42
// meaningOfLife'ın Int türünde olduğu çıkarımı yapılır
```

Benzer şekilde, kayan noktalı değişmez için bir tür belirtmezseniz, Swift bir `Double`:

```
let pi = 3.14159
// pi'nin Double türünde olduğu çıkarımı yapılır
```

Swift her zaman seçer `Double`(ziyade `Float`) kayan noktalı sayıların türünü çıkarırken.

Bir ifadede tamsayı ve kayan nokta değişmezlerini birleştirirseniz, bir tür `Double`bağlamdan çıkarılacaktır:

```
let anotherPi = 3 + 0.14159
// başka bir Pi'nin de Double türünde olduğu çıkarımı yapılır.
```

gerçek değeri `3` kendi içinde açık bir türü yoktur ve bu nedenle uygun bir çıktı türü `Double` eklemenin bir parçası olarak bir kayan nokta değişmezinin varlığından çıkarılır.

Sayısal Değişmezler

Tamsayı değişmezleri şu şekilde yazılabilir:

- Ön *ondalık* sayı
- Bir *ikili* sayı, bir `0b` önek
- bir *Sekizlik* sayı, bir `0o` önek
- bir *Onaltılık* sayı, bir `0x` önek

Bu tamsayı değişmezlerinin tümü ondalık bir değere sahiptir. `17`:

```
let decimalInteger = 17
let binaryInteger = 0b10001    // ikili gösterimde 17
let octalInteger = 0o21        // sekizli gösterimde 17
let hexadecimalInteger = 0x11  // onaltılık gösterimde 17
```

Kayan nokta değişmezleri ondalık (ön ek olmadan) veya onaltılık (bir `0x` önek). Ondalık noktanın her iki tarafında her zaman bir sayı (veya onaltılık sayı) olmalıdır. Ondalık şamandıralarda ayrıca *üs* büyük harf veya küçük harfle gösterilen `e`; onaltılık kayan noktalar, büyük harf veya küçük harfle gösterilen bir üs içermelidir `p`.

Üslü ondalık sayılar için `exp` ile çarpılır ^{exp} :

- `1.25e2` 1.25×10^{2} veya `125.0`.
- `1.25e-2` 1.25×10^{-2} veya `0.0125`.

Üslü onaltılık sayılar için `exp` ile çarpılır ^{exp} :

- `0xFp2` 15×2^{2} veya `60.0`.
- `0xFp-2` 15×2^{-2} veya `3.75`.

Bu kayan noktalı değişmezlerin tümü ondalık değere sahiptir. `12.1875`:

```
let decimalDouble = 12.1875
let exponentDouble = 1.21875e1
let hexadecimalDouble = 0xC.3p0
```

Sayısal değişmezler, okunmasını kolaylaştırmak için fazladan biçimlendirme içerebilir. Hem tam sayılar hem de kayan noktalar fazladan sıfırlarla doldurulabilir ve

okunabilirliğe yardımcı olmak için alt çizgiler içerebilir. Biçimlendirmenin hiçbir türü değişmezin altında yatan değeri etkilemez:

```
let paddedDouble = 000123.456
let oneMillion = 1\_000\_000
let justOverOneMillion = 1\_000\_000.000\_000\_1
```

Sayısal Tür Dönüşümü

Kullan `Int`Negatif olmadıkları bilinse bile kodunuzdaki tüm genel amaçlı tamsayı sabitleri ve değişkenler için yazın. Günlük durumlarda varsayılan tamsayı türünü kullanmak, tamsayı sabitlerinin ve değişkenlerin kodunuzda hemen birlikte çalışabileceği ve tamsayı değişmez değerleri için çıkarılan türle eşleşeceği anlamına gelir.

Diğer tamsayı türlerini, harici bir kaynaktan açıkça boyutlandırılmış veriler veya performans, bellek kullanımı veya diğer gerekli optimizasyon için yalnızca eldeki görev için özel olarak gerektiğinde kullanın. Bu durumlarda açıkça boyutlandırılmış türleri kullanmak, herhangi bir kaza sonucu oluşan değer taşmalarını yakalamaya yardımcı olur ve kullanılan verilerin doğasını örtük olarak belgeler.

Tamsayı Dönüştürme

Bir tamsayı sabitinde veya değişkeninde saklanabilecek sayı aralığı, her sayısal tür için farklıdır. Bir `Int8`sabit veya değişken arasında sayıları saklayabilir `-128`ve `127`, oysa bir `UInt8`sabit veya değişken arasında sayıları saklayabilir `0`ve `255`. Kodunuz derlendiğinde, sabit veya boyutlu bir tamsayı türündeki değişkene sığmayan bir sayı hata olarak bildirilir:

```
let cannotBeNegative: UInt8 = -1
// UInt8 negatif sayıları saklayamaz ve bu nedenle bu bir hata bildirir
let tooBig: Int8 = Int8.max + 1
// Int8, maksimum değerinden daha büyük bir sayı saklayamaz
// ve bu nedenle bu da bir hata bildirir
```

Her sayısal tür, farklı bir değer aralığı depolayabildiğinden, duruma göre sayısal tür dönüştürmeyi seçmelisiniz. Bu katılım yaklaşımı, gizli dönüştürme hatalarını önler ve kodunuzda tür dönüştürme niyetlerinin açık olmasına yardımcı olur.

Belirli bir sayı türünü diğerine dönüştürmek için, mevcut değerle istenen türden yeni bir sayı başlatırsınız. Aşağıdaki örnekte, sabit `twoThousand`tipte `UInt16`, sabit ise `one`tipte `UInt8`. Aynı türden olmadıkları için doğrudan birlikte eklenemezler. Bunun yerine, bu örnek çağırır `UInt16(one)`yeni bir tane oluşturmak için `UInt16`değeri ile başlatıldı `one`, ve bu değeri orijinalin yerine kullanır:

```
let twoThousand: UInt16 = 2_000
let one: UInt8 = 1
let twoThousandAndOne = twoThousand + UInt16(one)
```

Çünkü eklemenin her iki tarafı da artık tipte `UInt16`, eklemeye izin verilir. çıkış sabiti (`twoThousandAndOne`) türünde olduğu sonucuna varılır `UInt16`çünkü ikisinin toplamı `UInt16`değerler.

`SomeType(ofInitialValue)`Swift türünün başlatıcısını çağırmanın ve bir başlangıç değeri iletmenin varsayılan yoludur. Kamera ARKASI, `UInt16`kabul eden bir başlatıcıya sahip `UInt8`değer ve bu nedenle bu başlatıcı yeni bir değer oluşturmak için kullanılır. `UInt16`mevcut olandan `UInt8`. giremezsiniz *herhangi bir* Ancak burada `UInt16`bir başlatıcı sağlar. Yeni türleri kabul eden başlatıcılar sağlamak için mevcut türleri genişletme (kendi tür tanımlarınız dahil) Extensions .

Tamsayı ve Kayan Nokta Dönüşümü

Tamsayı ve kayan noktalı sayısal türleri arasındaki dönüşümler açık hale getirilmelidir:

```
let three = 3
let pointOneFourOneFiveNine = 0.14159
let pi = Double(three) + pointOneFourOneFiveNine
// pi, 3.14159'a eşittir ve Double türünde olduğu çıkarımı yapılır
```

Burada sabitin değeri `three`yeni bir tür değeri oluşturmak için kullanılır `Double`, böylece toplamanın her iki tarafı da aynı tipte olur. Bu dönüştürme yapılmadan, eklemeye izin verilmeyecektir.

Kayan noktadan tamsayıya dönüştürme de açık hale getirilmelidir. Bir tamsayı türü ile başlatılabilir `Double`veya `Float`değer:

```
let integerPi = Int(pi)
// integerPi 3'e eşittir ve Int türünde olduğu çıkarımı yapılır
```

Bu şekilde yeni bir tamsayı değeri başlatmak için kullanıldığında kayan nokta değerleri her zaman kesilir. Bu şu demek `4.75`olur `4`, ve `-3.9`olur `-3`.

Not: Sayısal sabitleri ve değişkenleri birleştirme kuralları, sayısal değişmezlere ilişkin kurallardan farklıdır. gerçek değer `3`doğrudan gerçek değere eklenebilir `0.14159`, çünkü sayı değişmezlerinin kendi içlerinde açık bir türü yoktur. Türleri yalnızca derleyici tarafından değerlendirildikleri noktada belirlenir.

Tür Takma Adları

Tür takma adları, mevcut bir tür için alternatif bir ad tanımlar. Tür takma adlarını şununla tanımlarsınız: `typealias`anahtar kelime.

Tür takma adları, örneğin harici bir kaynaktan belirli bir boyuttaki verilerle çalışırken olduğu gibi, mevcut bir türe bağlamsal olarak daha uygun bir adla başvurmak istediğinizde kullanışlıdır:

```
typealias AudioSample = UInt16
```

Bir tür takma adı tanımladığınızda, takma adı orijinal adı kullanabileceğiniz her yerde kullanabilirsiniz:

```
var maxAmplitudeFound = AudioSample.min
// maxAmplitudeFound şimdi 0'dır
```

Burada, `AudioSample` için bir takma ad olarak tanımlanır `UInt16`. Bir takma ad olduğu için, çağrı `AudioSample.min` aslında arar `UInt16.min` başlangıç değerini sağlayan `0` için `maxAmplitudeFound` değişken.

Booleans

Swift, adı verilen temel bir *Boole* türüne `Bool`. Boole değerleri *mantıksal* , çünkü yalnızca doğru veya yanlış olabilirler. Swift iki Boole sabit değeri sağlar, `true` ve `false`:

```
let orangesAreOrange = true
let turnipsAreDelicious = false
```

türleri `orangesAreOrange` ve `turnipsAreDelicious` olarak çıkarılmıştır `Bool` Boolean değişmez değerleriyle başlatıldıkları gerçeğinden. olduğu gibi `Int` ve `Double` yukarıda, sabitleri veya değişkenleri şu şekilde bildirmeniz gerekmez: `Bool` eğer onları ayarlarsan `true` veya `false` onları yaratır yaratmaz. Tür çıkarımı, türü zaten bilinen diğer değerlerle sabitleri veya değişkenleri başlattığında Swift kodunun daha özlü ve okunabilir olmasına yardımcı olur.

Boole değerleri, özellikle aşağıdaki gibi koşullu ifadelerle çalışırken kullanışlıdır: `if` ifade:

```
if turnipsAreDelicious {
print("Mmm, tasty turnips!")
} else {
print("Eww, turnips are horrible.")
}
// Prints "Eww, şalgamlar korkunç."
```

gibi koşullu ifadeler `if` daha ayrıntılı olarak ele alınmaktadır Kontrol Akışında .

Swift'in tür güvenliği, Boolean olmayan değerlerin yerine kullanılmasını engeller. `Bool`. Aşağıdaki örnek, bir derleme zamanı hatası bildirir:

```
let i = 1
if i {
// bu örnek derlenmeyecek ve bir hata bildirecek
}
```

Ancak, aşağıdaki alternatif örnek geçerlidir:

I

```
et i = 1
if i == 1 {
// bu örnek başarıyla derlenecek
}
```

sonucu `i == 1` karşılaştırma türdendir `Bool`, ve böylece bu ikinci örnek, tür denetimini geçer. gibi karşılaştırmalar `i == 1` bölümünde tartışılmaktadır Temel Operatörler .

Swift'deki diğer tür güvenliği örneklerinde olduğu gibi, bu yaklaşım tesadüfi hataları önler ve belirli bir kod bölümünün amacının her zaman açık olmasını sağlar.

Tuples

Tuples , birden çok değeri tek bir bileşik değerde gruplandırır. Bir tanımlama grubu içindeki değerler herhangi bir türden olabilir ve birbirleriyle aynı türden olmaları gerekmez.

Bu örnekte, `(404, "Not Found")` bir *HTTP durum kodunu* . HTTP durum kodu, bir web sayfası talep ettiğinizde bir web sunucusu tarafından döndürülen özel bir değerdir. Bir durum kodu `404 Not Found` var olmayan bir web sayfası talep ederseniz döndürülür.

```
let http404Error = (404, "Not Found")
// http404Error, (Int, String) türündedir ve eşittir (404, "Not Found")
```

bu `(404, "Not Found")` tuple grupları birlikte bir `Int` ve bir `String` HTTP durum koduna iki ayrı değer vermek için: bir sayı ve insan tarafından okunabilir bir açıklama. “Tip tipi bir demet” olarak tanımlanabilir. `(Int, String)` ”.

Türlerin herhangi bir permütasyonundan tanımlama grupları oluşturabilirsiniz ve bunlar istediğiniz kadar farklı tür içerebilir. Bir demet tipine sahip olmanızı engelleyen hiçbir şey yok `(Int, Int, Int)`, veya `(String, Bool)` veya gerçekten ihtiyacınız olan herhangi bir başka permütasyon.

Bir *demetin* içeriğini, daha sonra her zamanki gibi erişeceğiniz ayrı sabitlere veya değişkenlere ayırabilirsiniz:

```
let (statusCode, statusMessage) = http404Error
print("The status code is (statusCode)")
// Prints "The status code is 404"
print("The status message is (statusMessage)")
// Prints "The status message is Not Found"
```

Demetin yalnızca bazı değerlerine ihtiyacınız varsa, demetin alt çizgi () olan kısımlarını yoksayın.) demeti ayrıştırdığınızda:

```
let (justTheStatusCode, _) = http404Error
print("The status code is (justTheStatusCode)")
// Prints "The status code is 404"
```

Alternatif olarak, sıfırdan başlayan dizin numaralarını kullanarak bir tanımlama grubundaki bireysel öğe değerlerine erişin:

```
print("The status code is (http404Error.0)")
// Prints "The status code is 404"
print("The status message is (http404Error.1)")
// Prints "The status message is Not Found"
```

Tanımlama grubu tanımlandığında, tanımlama grubundaki tek tek öğeleri adlandırabilirsiniz:

```
let http200Status = (statusCode: 200, description: "OK")
```

Bir demet içindeki öğeleri adlandırırsanız, bu öğelerin değerlerine erişmek için öğe adlarını kullanabilirsiniz:

```
print("The status code is (http200Status.statusCode)")
// Prints "The status code is 200"
print("The status message is (http200Status.description)")
// Prints "The status message is OK"
```

Tuple'lar, fonksiyonların dönüş değerleri olarak özellikle kullanışlıdır. Bir web sayfasını almaya çalışan bir işlev, `(Int, String)` sayfa alımının başarısını veya başarısızlığını tanımlayan tanımlama grubu türü. İşlev, her biri farklı türde iki farklı değere sahip bir tanımlama grubu döndürerek, sonucu hakkında yalnızca tek bir türden tek bir değer döndürmesinden daha yararlı bilgiler sağlar. Daha fazla bilgi için, bkz Birden Çok Dönüş Değeri Olan İşlevler .

Not: Tuple'lar, ilgili değerlerden oluşan basit gruplar için kullanışlıdır. Karmaşık veri yapılarının oluşturulmasına uygun değiller. Veri yapınızın daha karmaşık olması muhtemelse, onu bir tanımlama grubu yerine bir sınıf veya yapı olarak modelleyin. Daha fazla bilgi için, bkz Yapılar ve Sınıflar .

Optionals

kullanırsınız *seçenekleri* Bir değerin bulunmadığı durumlarda İsteğe bağlı iki olasılığı temsil eder: Ya bir değer vardır *ve* bu değere erişmek için isteğe bağlı paketi açabilirsiniz ya da *hiç* değer yoktur.

Not: Seçenekler kavramı C veya Objective-C'de mevcut değildir. Objective-C'deki en yakın şey geri dönme yeteneğidir. `nil`aksi takdirde bir nesne döndürecek bir yöntemden `nil`"geçerli bir nesnenin yokluğu" anlamına gelir. Ancak bu yalnızca nesneler için geçerlidir; yapılar, temel C türleri veya numaralandırma değerleri için çalışmaz. Bu türler için Objective-C yöntemleri genellikle özel bir değer döndürür (örneğin, `NSNotFound`) bir değerin olmadığını belirtmek için. Bu yaklaşım, yöntemi çağıran kişinin test edilecek özel bir değer olduğunu bildiğini ve bunu kontrol etmeyi hatırladığını varsayar. Swift'in seçenekleri *herhangi bir tür* , özel sabitlere ihtiyaç duymadan

Burada, bir değerin yokluğuyla başa çıkmak için seçeneklerin nasıl kullanılabileceğine dair bir örnek verilmiştir. Swift'in `Int`type, bir dönüştürmeye çalışan bir başlatıcıya sahiptir. `String`değer `Int`değer. Ancak, her dize bir tamsayıya dönüştürülemez. dize `"123"`sayısal değere dönüştürülebilir `123`, ancak dize `"hello, world"`dönüştürülecek açık bir sayısal değere sahip değil.

Aşağıdaki örnek, bir dönüştürmeyi denemek için başlatıcıyı kullanır. `String`İçine `Int`:

```
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)
// convertNumber'ın "Int?" veya "optional Int" türünde olduğu anlaşılır.
```

Başlatıcı başarısız olabileceğinden, *isteğe bağlı* `Int`yerine, bir `Int`. isteğe bağlı `Int`olarak yazılır `Int?`, olumsuzluk `Int`. Soru işareti, içerdiği değerin isteğe bağlı olduğunu, yani *bazılarını* `Int`değer veya *hiç değer* . (Başka bir şey içeremez, örneğin `Bool`değer veya bir `String`değer. ya bir `Int`, ya da hiç bir şey değil.)

nil

İsteğe bağlı bir değişkeni, ona özel değer atayarak değersiz bir duruma ayarlarsınız. `nil`:

```
var serverResponseCode: Int? = 404
// serverResponseCode gerçek bir Int değeri olan 404 içerir
serverResponseCode = nil
// serverResponseCode artık değer içermiyor
```

Not: kullanamazsın `nil`isteğe bağlı olmayan sabitler ve değişkenlerle. Kodunuzdaki bir sabitin veya değişkenin belirli koşullar altında bir değerin yokluğunda çalışması

gerekiyorsa, bunu her zaman uygun türün isteğe bağlı bir değeri olarak bildirin.

Varsayılan bir değer sağlamadan isteğe bağlı bir değişken tanımlarsanız, değişken otomatik olarak `nil`senin için:

```
var surveyAnswer: String?
// surveyAnswer otomatik olarak sıfır olarak ayarlanır
```

Not: Swift'in `nil`aynı değil `nil`Objective-C'de. Objective-C'de, `nil`var olmayan bir nesneye işaretçidir. Swift'de, `nil`işaretçi değildir—belirli bir türde bir değerin olmamasıdır. isteğe bağlı seçenekler *Her* şu şekilde ayarlanabilir: `nil`, sadece nesne türleri değil.

If İfadeleri ve Zorla Açma

kullanabilirsiniz `if`İsteğe bağlı bir değer içerip içermediğini, isteğe bağlı ile karşılaştırarak bulmak için ifade `nil`. Bu karşılaştırmayı “eşittir” operatörü ile gerçekleştirirsiniz (`==`) veya "eşit değildir" operatörü (`!=`).

İsteğe bağlı bir değere sahipse, “eşit değil” olarak kabul edilir. `nil`:

```
if convertedNumber != nil {
print("convertedNumber contains some integer value.")
}
// Prints "convertedNumber contains some integer value."
```

emin olduğunuzda, *bir* bir ünlem işareti ekleyerek temel değerine erişebilirsiniz (`!`) isteğe bağlı adının sonuna. Ünlem işareti etkili bir şekilde şunu söylüyor: “Bu isteğe bağlı olarak kesinlikle bir değere sahip olduğunu biliyorum; lütfen kullanın." olarak bilinir *zorla açılması* isteğe bağlı değerin

```
if convertedNumber != nil {
print("convertedNumber has an integer value of (convertedNumber!).")
}
// Prints "convertedNumber has an integer value of 123."
```

hakkında daha fazlası için `if`deyimi, bkz Akış Kontrolü .

Not: Kullanmaya çalışıyorum `!` var olmayan bir isteğe bağlı değere erişmek için bir çalışma zamanı hatası tetiklenir. Her zaman bir isteğe bağlı öğenin aşağıdakileri içermediğinden emin olun: `nil`kullanmadan önce değer `!`değerini zorla-açmak için.

İsteğe Bağlı Bağlama

kullanırsınız *isteğe bağlı bağlamayı* bağlı bir değerin bir değer içerip içermediğini öğrenmek ve eğer öyleyse, bu değeri geçici bir sabit veya değişken olarak kullanılabilir

hale getirmek için İsteğe bağlı ciltleme ile kullanılabilir `if` ve `while` isteğe bağlı içindeki bir değeri kontrol etmek ve bu değeri tek bir eylemin parçası olarak bir sabite veya değişkene çıkarmak için ifadeler. `if` ve `while` daha ayrıntılı olarak açıklanmıştır Kontrol Akışında .

için isteğe bağlı bir bağlama yazın `if` açıklama şu şekilde:

```
if let constantName = someOptional {
statements
}
```

yeniden yazabilirsiniz `possibleNumber` örnek isteğe bağlı bağlamayı kullanmak için Seçenekler bölümünden

```
if let actualNumber = Int(possibleNumber) {
        print("The string \"(possibleNumber)\" has an integer value of
(actualNumber)")
} else {
        print("The string \"(possibleNumber)\" couldn't be converted to
an integer")
}
// Prints "The string "123" has an integer value of 123"
```

Bu kod şu şekilde okunabilir:

“Eğer isteğe bağlı `Int` tarafından iade edildi `Int(possibleNumber)` bir değer içeriyorsa, adında yeni bir sabit ayarlayın `actualNumber` isteğe bağlı olarak içerilen değere."

Dönüştürme başarılı olursa, `actualNumber` sabitinin ilk dalında kullanıma hazır hale gelir. `if` ifade. bulunan değerle zaten başlatıldı *içinde* ve bu nedenle `!` değerine erişmek için son ek. Bu örnekte, `actualNumber` sadece dönüşümün sonucunu yazdırmak için kullanılır.

İsteğe bağlı bağlama ile hem sabitleri hem de değişkenleri kullanabilirsiniz. değerini değiştirmek istiyorsanız `actualNumber` ilk şubesi içinde `if` açıklama yazabilirsin `if var actualNumber` bunun yerine, isteğe bağlı içinde yer alan değer, bir sabit yerine bir değişken olarak kullanılabilir hale getirilecektir.

Tek bir dosyaya istediğiniz kadar isteğe bağlı bağlama ve Boole koşulu ekleyebilirsiniz. `if` virgülle ayırarak ihtiyacınız olan ifadeyi kullanın. İsteğe bağlı bağlamalardaki değerlerden herhangi biri `nil` veya herhangi bir Boole koşulu şu şekilde değerlendirilir: `false`, bütün `if` ifadenin durumu olarak kabul edilir `false`. Devamındaki `if` ifadeler eşdeğerdir:

```
if let firstNumber = Int("4"), let secondNumber = Int("42"),
  firstNumber < secondNumber && secondNumber < 100
{
```

```
    print("(firstNumber) < (secondNumber) < 100")
}
// Prints "4 < 42 < 100"
if let firstNumber = Int("4") {
    if let secondNumber = Int("42") {
        if firstNumber < secondNumber && secondNumber < 100 {
            print("(firstNumber) < (secondNumber) < 100")
        }
    }
}
// Prints "4 < 42 < 100"
```

Not: İsteğe bağlı bağlama ile oluşturulan sabitler ve değişkenler `if` deyimi yalnızca gövdesinde mevcuttur `if` ifade. Buna karşılık, bir ile oluşturulan sabitler ve değişkenler `guard` deyimi izleyen kod satırlarında mevcuttur `guard` bölümünde açıklandığı gibi bildirim Erken Çıkış .

Dolaylı Olarak Açılmış Seçenekler

Yukarıda açıklandığı gibi, isteğe bağlı seçenekler, bir sabitin veya değişkenin "değersiz" olmasına izin verildiğini belirtir. Opsiyonel olarak kontrol edilebilir. `if` bir değerin olup olmadığını görmek için ifade ve isteğe bağlı değer varsa, isteğe bağlı değere erişmek için isteğe bağlı bağlama ile koşullu olarak açılabilir.

isteğe bağlı *her zaman* , bu değer ilk ayarlandıktan sonra Bu durumlarda, her erişildiğinde isteğe bağlı değerini kontrol etme ve paketini açma ihtiyacını ortadan kaldırmak yararlıdır, çünkü her zaman bir değere sahip olduğu güvenle varsayılabilir.

Bu tür isteğe bağlı seçenekler, *örtük olarak açılmamış isteğe* . Bir ünlem işareti koyarak (`String!`) soru işareti yerine (`String?`) isteğe bağlı yapmak istediğiniz türden sonra. Kullanırken isteğe bağlı adından sonra bir ünlem işareti koymak yerine, onu bildirirken isteğe bağlı türünden sonra bir ünlem işareti koyarsınız.

Örtülü olarak açılmamış seçenekler, bir isteğe bağlı değerinin, isteğe bağlı ilk tanımlandıktan hemen sonra var olduğu onaylandığında ve bundan sonraki her noktada kesinlikle var olduğu varsayıldığında yararlıdır. bölümünde açıklandığı gibi sınıf başlatma sırasındadır Unown References ve Implicitly Unwrapped İsteğe Bağlı Özellikler .

Örtülü olarak açılmamış bir isteğe bağlı, arka planda normal bir isteğe bağlıdır, ancak isteğe bağlı olmayan bir değer gibi, her erişildiğinde isteğe bağlı değeri açmaya gerek kalmadan da kullanılabilir. Aşağıdaki örnek, sarılmış değerlerine açık olarak erişirken, isteğe bağlı bir dize ile örtük olarak sarılmamış isteğe bağlı bir dize arasındaki davranış farkını gösterir. `String`:

```
let possibleString: String? = "An optional string."
let forcedString: String = possibleString! // requires an exclamation
```

```
point
let assumedString: String! = "An implicitly unwrapped optional string."
let implicitString: String = assumedString // no need for an exclamation
point
```

Örtülü olarak açılmamış isteğe bağlı bir seçeneği, isteğe bağlı olanın gerektiğinde zorla açılmasına izin vermek olarak düşünebilirsiniz. Örtülü olarak açılmamış isteğe bağlı bir değer kullandığınızda, Swift önce onu sıradan bir isteğe bağlı değer olarak kullanmaya çalışır; isteğe bağlı olarak kullanılamıyorsa, Swift değeri zorla açar. Yukarıdaki kodda, isteğe bağlı değer `assumedString`değerini atamadan önce zorla açılır `implicitString`çünkü `implicitString`açık, isteğe bağlı olmayan bir türü vardır `String`. Aşağıdaki kodda, `optionalString`açık bir türü yoktur, bu nedenle sıradan bir isteğe bağlıdır.

```
let optionalString = assumedString
// The type of optionalString is "String?" and assumedString isn't force-
unwrapped.
```

Örtülü olarak açılmamış bir isteğe bağlı ise `nil`ve sarılmış değerine erişmeye çalışırsanız, bir çalışma zamanı hatası tetiklersiniz. Sonuç, bir değer içermeyen normal bir isteğe bağlı sonra bir ünlem işareti koymanız ile tamamen aynıdır.

Örtülü olarak açılmamış bir isteğe bağlı olup olmadığını kontrol edebilirsiniz. `nil`normal bir isteğe bağlı kontrol ettiğiniz gibi:

```
if assumedString != nil {
print(assumedString!)
}
// Prints "An implicitly unwrapped optional string."
```

Değerini tek bir ifadede kontrol etmek ve sarmak için isteğe bağlı bağlama ile örtük olarak açılmamış isteğe bağlı bir seçenek de kullanabilirsiniz:

```
if let definiteString = assumedString {
print(definiteString)
}
// Prints "An implicitly unwrapped optional string."
```

Not: Bir değişken olma olasılığı olduğunda, örtük olarak açılmamış bir isteğe bağlı kullanmayın. `nil`daha sonraki bir noktada. olup olmadığını kontrol etmeniz gerekiyorsa, her zaman normal bir isteğe bağlı tür kullanın. `nil`Bir değişkenin ömrü boyunca değer.

Hata İşleme

kullanırsınız *hata işlemeyi* Programınızın yürütme sırasında karşılaşabileceği hata koşullarına yanıt vermek için

Bir işlevin başarısını veya başarısızlığını bildirmek için bir değerin varlığını veya yokluğunu kullanabilen seçeneklerin aksine, hata işleme, başarısızlığın altında yatan nedeni belirlemenize ve gerekirse hatayı programınızın başka bir bölümüne yaymanıza olanak tanır. .

Bir işlev bir hata koşuluyla karşılaştığında, bir hata *atar* . Bu işlevin arayanı daha sonra *yakalayabilir* hatayı

```
func canThrowAnError() throws {
// bu işlev bir hata verebilir veya vermeyebilir
}
```

Bir işlev, aşağıdakileri ekleyerek bir hata atabileceğini gösterir. `throws` beyanında anahtar kelime. Hata verebilecek bir işlevi çağırdığınızda, `try` ifadenin anahtar sözcüğü.

Swift, bir hata tarafından işlenene kadar hataları otomatik olarak mevcut kapsamlarının dışına yayar. `catch` madde.

```
do {
  try canThrowAnError()
  // hata atılmadı
} catch {
  // bir hata atıldı
}
```

A `do` ifadesi, hataların bir veya daha fazla alana yayılmasına izin veren yeni bir kapsam oluşturur. `catch` maddeleri.

Farklı hata koşullarına yanıt vermek için hata işlemenin nasıl kullanılabileceğine dair bir örnek:

```
func makeASandwich() throws {
  // ...
}
do {
  try makeASandwich()
  eatASandwich()
} catch SandwichError.outOfCleanDishes {
  washDishes()
} catch SandwichError.missingIngredients(let ingredients) {
  buyGroceries(ingredients)
}
```

Bu örnekte, `makeASandwich()` temiz tabak yoksa veya herhangi bir bileşen eksikse işlev hata verecektir. Çünkü `makeASandwich()` hata verebilir, işlev çağrısı bir `try` ifade. İşlev çağrısını bir `do` deyimi, atılan herhangi bir hata sağlanana yayılacaktır. `catch` maddeleri.

Herhangi bir hata atılmazsa, `eatASandwich()` fonksiyon denir. Bir hata atılırsa ve `SandwichError.outOfCleanDishes` durumda, daha sonra `washDishes()` fonksiyon çağrılır. Bir hata atılırsa ve `SandwichError.missingIngredients` durumda, daha sonra `buyGroceries(_:)` işlev ilişkili ile çağrılır `[String]` tarafından yakalanan değer `catch` model.

bölümünde daha ayrıntılı olarak Hata İşleme .

İddialar ve Ön Koşullar

İddialar ve *ön koşullar* , çalışma zamanında gerçekleşen kontrollerdir. Daha fazla kod çalıştırmadan önce temel bir koşulun karşılandığından emin olmak için bunları kullanırsınız. Onay veya ön koşuldaki Boole koşulu, `true`, kod yürütme her zamanki gibi devam eder. Koşul şu şekilde değerlendirilirse `false`, programın mevcut durumu geçersiz; kod yürütme sona erer ve uygulamanız sonlandırılır.

Kodlama sırasında yaptığınız varsayımları ve beklentilerinizi ifade etmek için iddiaları ve ön koşulları kullanırsınız, böylece bunları kodunuzun bir parçası olarak dahil edebilirsiniz. İddialar, geliştirme sırasında hataları ve yanlış varsayımları bulmanıza yardımcı olur ve ön koşullar, üretimdeki sorunları tespit etmenize yardımcı olur.

Çalışma zamanında beklentilerinizi doğrulamanın yanı sıra, iddialar ve ön koşullar da kod içinde yararlı bir belgeleme biçimi haline gelir. bölümünde tartışılan hata koşullarının aksine, Hata İşleme kurtarılabilir veya beklenen hatalar için iddialar ve ön koşullar kullanılmaz. Başarısız bir onay veya ön koşul, geçersiz bir program durumunu gösterdiğinden, başarısız bir iddiayı yakalamanın bir yolu yoktur.

İddiaları ve önkoşulları kullanmak, kodunuzu geçersiz koşulların ortaya çıkma olasılığı olmayacak şekilde tasarlamanın yerini tutmaz. Ancak, bunları geçerli verileri ve durumu zorlamak için kullanmak, geçersiz bir durum oluştuğunda uygulamanızın daha öngörülebilir şekilde sonlandırılmasına neden olur ve sorunun hata ayıklamasını kolaylaştırmaya yardımcı olur. Geçersiz bir durum tespit edilir edilmez yürütmenin durdurulması, bu geçersiz durumun neden olduğu hasarın sınırlandırılmasına da yardımcı olur.

İddialar ve ön koşullar arasındaki fark, kontrol edildiklerinde ortaya çıkar: İddialar yalnızca hata ayıklama yapılarında kontrol edilir, ancak ön koşullar hem hata ayıklama hem de üretim yapılarında kontrol edilir. Üretim yapılarında, bir iddianın içindeki koşul değerlendirilmez. Bu, geliştirme süreciniz sırasında üretimdeki performansı etkilemeden istediğiniz kadar iddia kullanabileceğiniz anlamına gelir.

İddialarla Hata Ayıklama

arayarak bir iddia yazarsınız. `assert(_:_:file:line:)` Swift standart kitaplığından işlev. Bu işlevi değerlendiren bir ifade iletirsiniz `true` veya `false` ve koşulun sonucu ise görüntülenecek bir mesaj `false`. Örneğin:

```
let age = -3
assert(age >= 0, "Bir kişinin yaşı sıfırdan küçük olamaz.")
// -3 >= 0 olmadığı için bu iddia başarısız olur.
```

Bu örnekte, aşağıdaki durumlarda kod yürütme devam eder: `age >= 0` değerlendirir `true`, yani, eğer değeri `age` negatif değil. değeri ise `age` negatif ise, yukarıdaki kodda olduğu gibi, o zaman `age >= 0` değerlendirir `false`, ve iddia başarısız olur ve uygulama sonlandırılır.

Onay mesajını atlayabilirsiniz - örneğin, koşulu düzyazı olarak tekrarladığında.

```
assert(age >= 0)
```

Kod durumu zaten kontrol ediyorsa, `assertionFailure(_:file:line:)` Bir iddianın başarısız olduğunu gösteren işlev. Örneğin:

```
if age > 10 {
  print("Hız trenine veya dönme dolaba binebilirsin.")
} else if age >= 0 {
  print("Dönme dolaba binebilirsiniz.")
} else {
  assertionFailure("Bir kişinin yaşı sıfırdan küçük olamaz.")
}
```

Ön Koşulların Uygulanması

Bir koşulun yanlış olma potansiyeli olduğunda, ancak *kesinlikle* kodunuzun yürütmeye devam etmesi için Örneğin, bir alt simgenin sınırların dışında olup olmadığını veya bir işleve geçerli bir değer iletildiğini kontrol etmek için bir ön koşul kullanın.

arayarak bir ön koşul yazarsınız. `precondition(_:_:file:line:)` işlev. Bu işlevi değerlendiren bir ifade iletirsiniz `true` veya `false` ve koşulun sonucu ise görüntülenecek bir mesaj `false`. Örneğin:

```
// Bir aboneliğin uygulanmasında...
precondition(index > 0, "Dizin sıfırdan büyük olmalıdır.")
```

ayrıca arayabilirsiniz `preconditionFailure(_:file:line:)` örneğin, bir anahtarın varsayılan durumu alınmışsa, ancak tüm geçerli giriş verileri anahtarın diğer durumlarından biri tarafından işlenmiş olmalıdır.

Not: İşaretlenmemiş modda derlerseniz (`-Ounchecked`), ön koşullar kontrol edilmez. Derleyici, ön koşulların her zaman doğru olduğunu varsayar ve kodunuzu buna göre optimize eder. Ancak `fatalError(_:file:line:)` işlevi, optimizasyon ayarlarından bağımsız olarak yürütmeyi her zaman durdurur.

kullanabilirsiniz `fatalError(_:file:line:)` Prototip oluşturma ve erken geliştirme sırasında henüz uygulanmayan işlevsellik için taslaklar oluşturmak için yazarak, `fatalError("Unimplemented")` saplama uygulaması olarak. Önermelerin veya ön koşulların aksine, önemli hatalar hiçbir zaman optimize edilmediğinden, bir saplama uygulamasıyla karşılaşırsa yürütmenin her zaman durduğundan emin olabilirsiniz.