

Swift Programlama Dili

Kurs Notları

Kaan ASLAN

C ve Sistem Programcıları Derneği

Güncelleme Tarihi: 14/12/2016

Bu kurs notları Kaan ASLAN tarafından yazılmıştır. Kaynak belirtilmek koşuluyla her türlü alıntı yapılabilir.

Apple Firmasının Kısa Tarihi

Apple şirketinin kurucuları Steve Jobs ve Steve Wozniak lisede (high school) sınıf arkadaşlarıydılar. Okul sonrası iletişimleri bir süre koptu. 1974 yılında Intel 8080'i çıkartıp ilk mikrobilgisayar olan Altair'i de yapıncı Amerika'da mikrobilgisayarlara ilgi çok arttı. Her yerde bilgisayar kulüpleri kuruldu. Genç girişimciler de çeşitli olanaklar için fırsat kollamaya başladılar. Bill Gates ve arkadaşı Paul Allen Altair için Basic yorumlayıcısı yazıp işi biraz büyötmüşlerdi. 1975 yılında Harvard'tan ayrılarak Microsoft şirketini kurdular.

Jobs ile Wozniak'ın ilişkileri okuldan sonra bir süre koptu. Steve Wozniak da bu atmosfer içerisinde bir bilgisayar yapma hevesine kapıldı. Intel'in 8080'inden sonra Motorola firması 6800 işlemcisi çıkartmıştı. Wozniak Altair'den de etkilenecek ilk mikrobilgisayarını yapmaya çalıştı. İşlemci olarak Intel'in 8080'ini değil, Motorola'nın 6800'ünü tercih etti. Wozniak bu arada Rockwell'in 6502 işlemcileri için de bir BASIC yorumlayıcısı yazdı. Bu sıralarda Wozniak bir bilgisayar kulübünde eski arkadaşı Jobs ile karşılaştı. Mikrobilgisayarlardan gelecek bekleyen ikili birlikte çalışma kararı aldılar.

1976 yılında Steve Jobs ve Steve Wozniak kafa kafaya vererek Jobs'ların garajında ilk Apple bilgisayarını derme çatma olarak yaptılar. Bu bilgisayarlar daha sonra geliştirilerek Apple 1 ismini aldı. Steve Jobs teknik biri değildi. Jobs daha çok iş geliştirme üzerine odaklanmıştı. Asıl mühendislik faaliyetlerini Wozniak üstlenmişti. Daha sonra bu ikiliye Ronald Wayne de katıldı. Ancak Wayne kısa süre sonra hisselerini diğer iki kişiye satarak gruptan ayrıldı. Apple 1 bilgisayarlarını 1977'de Apple 2 izledi. 1980 yılında Apple 3 piyasaya çıktı. Aynı yılın sonlarına doğru da IBM bugünkü PC'lerin atası olan IBM PC'yi piyasaya çıkarttı. IBM bu ilk PC'sinin işletim sistemini kendisi yazmayı –küçük iş olduğu gerekçesiyle- uygun görmemiş onu Microsoft isimli küçük bir firmaya yazdırmıştır. (O zamanlar Microsoft'ta 8 kişi çalışıyordu). IBM PC'nin çıkmasıyla Apple'ın küçük olan kişisel bilgisayar pazarındaki yeri geriye itilmeye başladı.

Apple 1983 yılında Lisa modelini çıkardı. 10000\$'a yakın bir fiyatı vardı Lisa'nın. Bu nedenle satışı da fazla olmadı. Lisa ilk fare ve GUI kullanan işletim sistemiydi. 1983 yılının Aralık ayında ilk Machintosh serisi bilgisayar piyasaya süröldü. Bunun ismi "Machintosh 128K" idi. Apple daha sonraları Machintosh ismi yerine Mac ismini kullanmaya başladı. Bunu 1984 yılında Machintosh 512K izledi. Machintosh'lar grafik arayüzleri nedeniyle özellikle masaüstü yayıncılık işinde kendilerine bir yer buldular. Ancak satışları beklenildiği gibi olmadı.

Machintosh satışları çok da yüksek olmayınca Apple şirketinde zaten var olan çatışmalar hepten kendini gösterdi. Bu çatışmalar 1985 yılında Steve Jobs'un şirketi terk etmesiyle sonuçlandı. Jobs Apple'dan kovulunca NeXT isimli bir şirket kurdu. NeXT şirketi 1985 yılında NeXT bilgisayarlarını piyasaya sürmüştür. NeXT bilgisayarlarında NeXTSTEP isimli işletim sistemi kullanılıyordu. Daha sonra bu sistem açık hale getirildi ve OPENSTEP ismini aldı. Dünyanın ilk Web tarayıcısı Tim Berners Lee tarafından Cern'de NeXT bilgisayarları üzerinde gerçekleştirilmiştir.

Jobs NeXT firmasını kurup oarada işlerine devam ederken Apple 1987'de Machintosh 2'yi piyasaya çıkardı. Microsoft'ta 1985 yılında yavaş yavaş GUI arayüzü uygulamasına geçmeye başlamıştır. Microsoft Windows'un ilk versiyonu 1985 yılında çıkardı. Bunu daha sonra 2.0 versiyonu, sonra 3.0 versiyonu ve sonra da 3.1 versiyonu izledi. Ancak Microsoft'un bu 16 bit Windows sistemleri birer işletim sistemi değildi. DOS'tan çalıştırılan bir çeşit utility program gibiydiler. Apple firması GUI arayüzünü Mac'lerden çaldığı gerekçesiyle Microsoft'u mahkemeye verdi. Ancak mahkemeyi kazanamadı.

Apple 1990'da "Machintosh Classic" isimli modelini çıkardı. Apple işlemci olarak Motorola 68000 ailesini bırakarak Power PC ailesine geçti.

Steve Jobs 1997 yılında Apple'a geri döndü. Apple da NeXT firmasını 200 milyon dolara satın aldı. Sonra piyasaya iMac ve Power Mac serileri çıktı. Daha sonra Steve Jobs Mac'lerin çekirdeklerini tamamen değiştirme kararı aldı. Mac'ler Mac OS 10 ile birlikte yeni bir çekirdeğe geçtiler. Apple İpod'lar ile müzik aygıtları piyasasında bir yer edindi. 2005 yılında Apple Mac'lerin işlemcilerini de değiştirme kararı aldı Böylece Power PC ailesi bırakılarak Intel ailesine geçildi. 2007 yılında Iphone'lar ve 2010 yılında ilk İpad'ler piyasaya çıktı. Bu ürünler bir satış grafiği ile Apple'a çok para kazandırdı. Apple'ın geliri 2010'dan itibaren iyice arttı ve dünyanın en büyük IT firması durumuna geldi.

Machintosh Bilgisayarlarında Kullanılan İşletim Sistemleri

1984 yılındaki ilk Machintosh'un işletim sistemi Mac OS'ti. Mac OS monolitik bir çekirdeğe sahipti ve MFS (Machintosh File System) isimli bir dosya sistemi kullanıyordu. 1985 yılında Machintosh'ların dosya sistemi HFS (Hierarchical File System) olarak değiştirildi. 2000'lerin başlarında Mac OS X ile birlikte Mac OS sistemlerinin çekirdekleri tamamen değiştirildi.

Mac OS X UNIX türevi bir işletim sistemidir. Çekirdeğine Darwin denilmektedir. Darwin açık bir sistemdir. Ancak Mac OS X tam anlamıyla açık bir sistem değildir. Yani MAC OS X, Darwin çekirdeğini kullanan GUI arayüzleri olan telif bir sistemdir.

Darwin'in hikayesi 1989 yılında NeXT'in NeXTSTEP işletim sistemiyle başladı. NeXTSTEP daha sonra OPENSTEP oldu. Apple NeXT'i 1996'nın sonunda 1997'nin başında satın aldı ve sonraki işletim sistemini OPENSTEP üzerine kuracağını açıkladı. Bundan sonra Apple 1997'de OPENSTEP üzerine kurulu olan Rhapsody'yi çıkardı. 1998'de de yeni işletim sisteminin Mac OS X olacağını açıkladı. Daha sonra Rhapsody'den Darwin projesi türedi. Darwin projesi ayrı bir işletim sistemi olarak da yüklenebilmektedir. Ancak Darwin grafik arayüzü olmadığı için Mac programlarını çalıştıramaz.

Darwin'den çeşitli projeler türetilmiştir. Bunlardan biri Apple tarafından 2002'de başlatılan OpenDarwin'dir. Bu proje 2006'da sonlandırılmıştır. 2007'de PureDarwin projesi başlatılmıştır.

Darwin'in çekirdeği XNU üzerine oturtulmuştur. XNU bir çekirdektir ve NeXT firması tarafından NEXTSTEP işletim sisteminde kullanılmak üzere geliştirilmiştir. XNU, Carnegie Mellon (Karnegi diye okunuyor) üniversitesi'nin Mach 3 mikrokernel çekirdeği ile 4.3BSD karışımı hibrit bir sistemdir.

Mac OS X sistemlerinin versiyonları şunlardır:

- Mac OS X 10.0 (Cheetah, 2001)
- Mac OS X 10.1 (Puma, 2001)
- Mac OS X 10.2 (Jaguar, 2002)
- Mac OS X 10.3 (Panther, 2003)
- Mac OS X 10.4 (Tiger, 2005)
- Mac OS X 10.5 (Leopard, 2007)
- Mac OS X 10.6 (Snow Leopard, 2009)
- Mac OS X 10.7 (Lion, 2011)
- Mac OS X 10.8 (Mountain Lion, 2012)
- Mac OS X 10.9 (Mavericks, 2013)
- Mac OS X 10.10 (Yosemite, 2014)
- Mac OS X 10.11 (El Capitan, 2015)
- Mac OS X 10.12 (Sierra, 2017)
- Mac OS X 10.12 (High Sierra, 2017)

Mac OS X sistemlerinde dosya sistemi olarak HFS'nin sonraki sürümü olan HFS+ kullanılmaktadır.

IOS İşletim Sistemlerinin Tarihsel Gelişimi

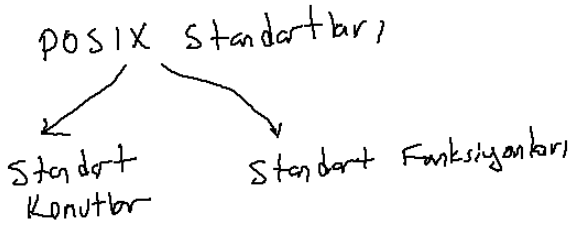
IOS ismi "Iphone Operating System"dan kısaltmadır. Bugün için IOS işletim sistemlerinin akıllı telefon ve tabletlerdeki kullanım oranları %20 civarındadır. IOS İşletim sistemi Apple'ın Iphone, Ipad, Ipod, IWatch ve AppleTV cihazlarında kullanılmaktadır. IOS adeta Mac OS X'in mobil versiyonu gibidir. Mac ile IOS sistemlerinde benzer araçlar ve ortamlarla (frameworks) uygulama geliştirilmektedir. IOS'ta büyük ölçüde Darwin çekirdeğinin kodları kullanılmıştır. IOS işletim sistemlerinin tarihsel gelişimi şöyledir:

- iPhone OS 1 (2007)
- iPhone OS 2 (2008)
- iPhone OS 3 (2009)
- IOS 4 (2010)
- IOS 5 (2012)
- IOS 6 (2012)
- IOS 7 (2013)
- IOS 8 (2014)
- IOS 9 (2015)
- IOS 10 (2016)
- IOS 11 (2017)

Mac Bilgisayarlarının Kullanımı Üzerine Anahtar Bazı Bilgiler

- Mac'lerde her programın ayrı bir menü çubuğu yoktur. Tek bir menü çubuğu vardır. Programlar aktive edilince menü çubuğunun içeriği de değişmektedir. (Bu tasarımda menü çubukları toplamda daha az yer kaplamış oluyor)

- Mac bilgisayarlarındaki fare geleneksel olarak tek tuşludur. Ancak bunlara PC'lerdeki fareler bağlanabilir bu durumda sağ tuş da kullanılabilir. Tek tuşlu farede sağ tuş etkisi yaratmak için Control + Click yapılır.
- Mac'lerde Del tuşu yoktur. Del etkisi yaratmak için fn + Backspace ya da Cmd + BackSpace kullanılır.
- Mac'lerde Page Up , Page Down, Home ve End tuşları yoktur. Bunların yerine fn ve ok tuşları kullanılır.
- Mac'lerdeki kombine tuşları şunlardır:
Command ⌘ Shift ⇧ Option ⌥ Control ^ Caps Lock ⌵ Fn
- Cmd (Command) tuşu PC'lerdeki Control tuşuna yakın bir işleve sahiptir. Örneğin Cut, Copy, Paste için Cmd + X, Cmd + C ve Cmd + V tuşları kullanılır.
- Yeni Mac dizi üstü bilgisayarlara track pad'ler vardır. Bunlarda çok dokunuşlu (multitouch) el mimikleri (gestures) uygulanabilmektedir. Ayrıca Mac'ler için "Magic Mouse"lar da piyasaya sürülmüştür. Track Pad'teki el mimikleri şunlardır:
- Program icon'unu masaüstüne yerleştirip tıklamak yerine Mac'lerde Cmd + Backspace ile Spotlight yaygın olarak tercih edilmektedir.
- Farenin sağ tuşuna tıklama (Apple terminolojisine göre secondary click) iki parmakla tıklama ile yapılır. (Ya da kntrl + tek tıklama)
- Yukarı ve aşağı kaydırma için iki parmak yukarı aşağı hareket ettirilir.
- Tarayıcılarda olduğu gibi önceki sayfa ve sonraki sayfaya gitmek için iki parmak sola sağa hareket ettirilir.
- Zoom In ve Zoom Out için iki parmak açılır ve kapatılır (pinch işlemi, mobil aygıtlarda olduğu gibi)
- fn + 11 masaüstünü görüntülemekte kullanılır.
- Pencere taşıma için Tıklama + sürükleme işlemi yapılır.
- Resimleri iki parmak hareketiyle döndürülebilmektedir.
- Alttaki görev çubuğunun sağında minimize edilmiş ana pencereler bulunur. Sol tarafta uygulama listesi vardır (bunlar değiştirilebilir.) Buradaki uygulamaların altında siyah nokta olanlar çalışan uygulamaları belirtmektedir.
- Finder Windows sistemlerindeki Windows Explorer gibidir.
- Pencere icon'ları Windows sistemlerinin tersine pencerenin sol tarafında bulunmaktadır.
- Mac OS X sistemleri Snow Leopard'la birlikte tam POSIX uyumuna kavuşmuştur. Yani komut satırına (terminal) geçildiğinde UNIX/Linux sistemlerindeki standart komutlar buradan uygulanabilir. Yine biz Mac OS X sistemlerinde C programlama dilinde çalışıyorsak POSIX fonksiyonlarını (open, read, write, fork, ...) buradan kullanabiliriz.



- Mac OS X sistemlerinde kullanıcılar masaüstüne çok fazla şey yerleştirme gereksinimi duymazlar. Programlar genellikle "Uygulamalar" klasörünün içerisinde ve Spotlight Search (Cmd + Space) ile aranarak çalıştırılırlar.

Swift Programlama Dilinin Tarihi

Apple Steve Jobs'un eski şirketinde kullanılan Objective-C'yi ve Cocoa kütüphanesini Mac sistemlerinin temel geliştirme araçları haline getirdi. Böylece gerek Mac OS X sistemlerinde gerekse IOS sistemlerinde uygulamalar Objective-C dili kullanılarak Cocoa Framework'ü ile geliştiriliyordu. Objective-C klasik C programlama dilinin üzerine nesne yönelimli özellikler eklenerek oluşturulmuş bir dildir. Adeta C ile Smalltalk'un birleşimi gibi düşünülebilir. Objective-C'nin öğrenilmesi, önce C'nin öğrenilmesini gerektirdiği için zordur. İşte Apple bunu fark etmiş ve tıpkı Java gibi C# gibi kendi ortamları için daha kolay öğrenilebilecek modern bir programlama dili arayışına girmiştir. Bunun sonucunda Swift doğmuştur.

Swift çok yeni bir dildir. Swift'in geliştirilmesine Chris Lattner tarafından 2010 yılında başlandı. Sonra geliştirme ekibine başka programcılar da katıldı. Swift'in tasarım sürecinde pek çok dilin özellikleri incelenmiş ve pek çok dilden alıntılar yapılmıştır. Dilin beta versiyonu 2014 yılında piyasaya sürüldü. Sonra bunu 2014 yılının Eylül ayında ilk gerçek sürümü olan Swift 1.0 izledi.

Kasım 2014'te Swift'in 4.1'inci sürümü çıktı. Bunu da 2015 yılında Swift 2.0 izledi. Kursun verildiği tarih dikkate alındığında Swift'in en son sürümü 4.1'dir. Bu son sürüm Aralık 2015'te (yani içinde bulunduğumuz ay) kullanıma sokulmuştur. Swift evrimin evrimini tam tamamlamış bir programlama dili değildir. Şu anda Swift'in 3.0 uyarlaması üzerinde çalışılmaktadır. Zamanla Swift'te daha pek çok özellikler eklenecek ve mevcut özelliklerde bazı değişiklikler yapılacaktır. Ancak dilin temel özelliklerinin sabit kalacağı gelişmenin yeni özellik ekleme yoluyla yapılacağı düşünülmektedir.

Swift Nasıl Bir Dildir?

Swift çok modelli (multi paradigm) bir programlama dilidir. Swift ile nesne yönelimli (object oriented), fonksiyonel (functional) ve prosedürel (procedural) programlama modelleri kullanılabilir. Swift Objective-C'den pek çok özellik almıştır. Yani belli ölçülerde Objective-C'ye benzemektedir. Adeta Apple'ın kendi deyişiyle "C'siz bir Objective-C"yi andırmaktadır. Swift'te C'deki gibi gösterici kullanımı yoktur. İsimli parametreler gibi, protokoller gibi, kategoriler gibi Objective-C özellikleri Swift'te de bulundurulmuştur.

Swift başta Objective-C olmak üzere C#, Java, Python ve Haskell gibi dillerin pek çok güçlü özelliğini almıştır. Bu nedenle bu programlama dillerinden Swift'e geçenler pek çok tanıdık öğeyle karşılaşabilmektedir.

Swift temel olarak derleyicilerle çalışılan bir dildir. Swift derleyicileri gerçek makine kodları (native codes) üretmektedir. Bu nedenle Swift programları hızlı çalışır. Apple'ın Swift derleyicisi LLVM isimli derleyici geliştirme ortamından faydalanılarak yazılmıştır. Swift ile yazılan kodlar LLDB isimli debugger ile debug edilebilmektedir.

Swift'te Merhaba Dünya Programı

Ekrana "Merhaba Dünya" yazısını çıkaran bir Swift programı herhangi bir text editör kullanılarak yazılabilir. Text editör olarak Mac'in standart TextEdit'i biraz zahmet verebilir. Çünkü TextEdit temelde rich text editördür. (Windows'taki WordPad gibi). Bu da başlangıçta biraz karışıklığına yol açabilir. Eğer TextEdit kullanacaksanız iki şeye dikkat etmelisiniz:

- 1) TextEdit/Tercihler menüsünden "Düz Metin"i seçin.
- 2) Tırnakların normal ASCII tırnak olarak ele alınması için Sistem Tercihleri / Klavye / Metin kısmından gerekli değişikliği yapın.

Text editör olarak BlueFish gibi, JEdit gibi açık kaynak kodlu bir editör kullanabilirsiniz. Ya da doğrudan XCode'u da hiç proje yaratmadan editör olarak kullanabilirsiniz. Merhaba Dünya programı şöyle yazılabilir:

```
// sample.swift  
  
print("Merhaba Dünya")
```

Swift programlarının uzantısı geleneksel olarak .swift biçimindedir.

Programı derlemek için komut satırına geçilip şu komut uygulanır:

```
swiftc -o sample sample.swift
```

Eğer -o seçeneği belirtilmezse çalıştırılabilen dosyanın (executable file) ismi program isminin aynısı olur. (Yani örneğimizde "sample") UNIX/Linux sistemlerinde olduğu gibi çalıştırılabilen dosyanın bir uzantısı olmak zorunda değildir.

Programı çalıştırmak için kabuk üzerinde:

```
./sample
```

komutunu uygulayabilirsiniz. Mac OS X'in UNIX türevi bir işletim sistemi olduğunu unutmayınız. Bu nedenle komut satırında standart UNIX komutlarını uygulayabilirsiniz.

swift isimli program default durumda bize bir komut satırı ortamı sunmaktadır. Bu komut satırında biz tek tek komutları girip o anda onların sonuçlarını alabiliriz. Son zamanlarda yeni dillerin çoğu aynı zamanda böyle bir komut satırı da sunmaktadır. Bu komut satırına Swift dünyasında REPL (Read+Evaluate+Print+Run) denilmektedir. REPL denemeler için kullanılabilen karşılıklı etkileşimli bir ortamdır.

swift isimli program aynı zamanda Swift programlarının bir script gibi (yani yorumlayıcı gibi) çalışmasına da izin vermektedir. Biz swift isminin yanına bir dosya ismi verirsek swift programı bu dosyadaki Swift kodlarını o anda sanki bir yorumlayıcı gibi hiç hedef kod üretmeden çalıştırmaktadır. Örneğin:

swift sample.swift

Bu durumda bir Swift kodları üç biçimde çalıştırılabilmektedir:

- 1) swiftc isimli derleyici ile derleyerek çalıştırılabilir dosya elde etmek
- 2) swift isimli programla yorumlayıcı tarzda (arkada aslında yine bir derleme işlemi yapılmaktadır) doğrudan programı çalıştırmak.
- 3) swift programında dosya ismi vermeyip karşılıklı etkileşimli REPL ortamına girmek ve orada kodları tek tek yazmak.

Merhaba Dünya Programının Açıklaması

Merhaba Dünya programındaki print fonksiyonu Swift'in standart kütüphanesine ilişkin bir fonksiyondur. Swift'in pek çok fonksiyonu barındıran standart bir kütüphanesi vardır. Swift yukarıda da sözü edildiği gibi çok modelli (multi paradigm) bir progmlama dilidir. Dolayısıyla biz onu prosedürel biçimde (yani sınıf olmadan fonksiyonlarla) kullanabiliriz.

Peki Swift programları nereden çalışmaya başlar? Pek çok dilde main isminde bir başlangıç fonksiyonu (entry point function) bulunmaktadır. Fakat Swift'te programın çalışmaya başladığı böyle bir fonksiyon yoktur. Swift'te programlar kaynak kodun tepesinden çalışmaya başlarlar. (Bu konuda bazı ayrıntılar var. İleride ele alınacaktır.) Dolayısıyla Merhaba Dünya programında akış kaynak dosyanın tepesinden başlamaktadır. Orada da print fonksiyonu çağırılmıştır.

Dil Nedir?

Dil karmaşık bir olgudur. Bu nedenle tek bir cümleyle tanımını yapmak zordur. Ancak "iletişimde kullanılan semboller kümesi" olarak tanımlanabilir. Diller doğal diller ve kurgusal diller olmak üzere ikiye ayrılmaktadır. Doğal diller uzun yaşantılar sonucunda oluşmuşlardır. Örneğin Türkçe gibi, İngilizce gibi. Kurgusal diller ise insanlar tarafından belli bir amaç ve mantık çerçevesinde oluşturulmuşlardır. Bilgisayar dilleri kurgusal diller ailesindedir.

Bir dilin tüm kurallarına gramer denilmektedir. Gramerin de iki önemli alt alanı vardır: Sentaks ve semantik. Bu iki olgunun tüm dillerde mutlaka bulunması gerekir. Sentaks dili oluşturan en yalın öğelerin (atom) doğru ve geçerli bir biçimde bir aaraya getirilmesine ilişkin kurallardır. Örneğin:

I going am School to

Burada bir sentaks hatası yapılmıştır. Örneğin:

```
if a > b
    print("Ok")
```

Buradaki Swift kod parçasında da bir sentaks hatası yapılmıştır. Semantik ise doğru yazılmış ve dizilmiş atomların ne anlam ifade ettiğine ilişkin kurallardır. Örneğin if deyiminin nasıl yazılması gerektiği sentaksa ilişkin bir konu iken nasıl çalıştığı semantiğe ilişkin bir konudur. Bir olgunun dil olarak betimlenmesi için sentaks ve semantik kurallara sahip olması gerekir.

Bilgisayar dünyasında kullanılan kurgusal dillere "bilgisayar dilleri (computer languages)" denilmektedir. Bir bilgisayar dilinde bir akış da varsa buna "programlama dili (programming language)" denir. Örneğin XML bir bilgisayar dilidir ancak programlama dili değildir.

Atom (Token) Kavramı

Bir programlama dilindeki daha fazla parçaya bölünemeyen kendi başına anlamlı olan en küçük birime atom (token) denilmektedir. Program aslında atomların belli bir kurala göre bir araya getirilmesiyle oluşturulmaktadır. Atomlar doğal dillerdeki sözcüklere benzetilebilirler. Derleyiciler de derleme işleminin ilk aşamasında kaynak kodu atomlarına ayırmaktadır. Örneğin aşağıdaki swift programını dikkate alalım:

```
let a = 10
let b = 20
let c: Int

c = a + b
print(c)
```

Bu Swift programının atomları şunlardır:

```
let
a
=
10
let
b
=
20
let
c
:
Int
c
=
a
+
b
print
(
c
)
```

Atomlar 6 gruba ayrılmaktadır:

- 1) Anahtar Sözcükler (keywords / reserved words): Dil için özel anlamı olan değişken olarak kullanılması yasaklanmış sözcüklere anahtar sözcük denilmektedir. Örneğin if, for, while gibi atomlar birer anahtar sözcüktür.
- 2) Değişkenler (identifiers / variables): İsmi bizim ya da başkalarının istediği gibi verebildiği sözcüklerdir. Örneğin x, y, count, print birer değişken atomdur.
- 3) Operatörler (Operators): Bir işleme yol açan işlem sonucunda bir değer üretilmesini sağlayan atomlara operatör denir. Örneğin +, -, / gibi atomlar birer operatördür.

4) Sabitler (literals / constants): Doğrudan yazılan sayılara sabit denir. Örneğin 100, 200 gibi.

5) Stringler (strings): İki tırnak içerisinde yazılan yazılar iki tırnaklarıyla birlikte tek bir atomdur. Bunlara string denir.

6) Ayıraçlar (delimiters / punctuators): Yukarıdaki atom gruplarının dışında kalan ';' gibi , '{' gibi atomlara ayıraç atom denilmektedir.

Boşluk Karakterleri (White Space)

Klavyeden boşluk duygusu oluşturmak için kullanılan karakterlere boşluk karakterleri denir. Örneğin SPACE, TAB, ENTER, VTAB tipik boşluk karakterleridir. Swift'te de diğer pek çok dilde olduğu gibi boşluk karakterleri atom ayırıcı (token delimiter) olarak kullanılmaktadır.

Swift'in Yazım Kuralı

Swift'te de diğer pek çok modern dillerde olduğu gibi boşluk karakterleri atom ayırıcı olarak kullanılmaktadır. Atomlar arasında istenildiği kadar boşluk karakteri bulundurulabilir. Atomlar istenildiği kadar bitişik yazılabilirler. Fakat anahtar sözcüklerle değişken atomlar peşi sıra geliyorsa onların aralarına en az bir boşluk karakteri yerleştirmek gerekir.

Ancak Swift'te deyimlerdeki sonlandırıcı (terminator) karakter konusunda küçük bir farklılık vardır. Eğer aynı satır üzerinde birden fazla deyim bulunduruluyorsa (önceki deyimlerin son kısımları da dahil) bu durumda son deyimden önceki deyimlerin ';' atomu ile sonlandırılması gerekir. Örneğin:

```
var a =  
    10 a = 20          // geçersiz!
```

Fakat:

```
var a =  
    10; a = 20         // geçerli
```

Örneğin:

```
a = 10 b = 20          // geçersiz!
```

Fakat:

```
a = 10; b = 20         // geçerli
```

Örneğin:

```
if a > 10 {  
    total += 1  
}  
else {  
    total -= 1
```

```
} print(total)    // geçersiz!
```

Fakat:

```
if a > 10 {  
    total += 1  
}  
else {  
    total -= 1  
}; print(total)    // geçerli
```

Swift'in Temel Veri Türleri

Tür bir değişkenin bellekte kaç byte yer kapladığını, ona hangi aralıkta ve formatta değerler yerleştirilebileceğini ve onun hangi işlemlere sokulabileceğini belirten önemli bir özelliktir. Swift statik ve katı bir tür kontrol sistemine sahip bir programlama dilidir (strongly typed language). Swift'in temel veri türleri (tıpkı C#'ta olduğu gibi) yapı biçiminde organize edilmiştir. Aşağıdaki listede temel türler görülmektedir:

Tür Belirten Sözcük	Uzunluk (Byte)	Sınır Değerler
Int	Sistem bağımlı 4 ya da 8	[-2147483648, +2147483647] [-9223372036854775808, +9223372036854775807]
UInt	Sistem bağımlı 4 ya da 8	[0, 4294967295] [0, +18446744073709551615]
Int8	1	[-128, +127]
UInt8	1	[0, +255]
Int16	2	[-32768, +32767]
UInt16	2	[0, 65535]
Int32	4	[-2147483648, +2147483647]
UInt32	4	[0, 4294967295]
Int64	8	[-9223372036854775808, +9223372036854775807]
UInt64	8	[0, +18446744073709551615]
Float	4	[+3.6 * 10 ⁺³⁸ , +3.6 * 10 ⁻³⁸]
Double	8	[+1.8 * 10 ⁺³⁰⁸ , +1.8 * 10 ⁻³⁰⁸]
Character	2	UNICODE karakterlerin sıra numarası
Bool	1	true, false
String	Yapısal temsil	Birden fazla karakteri (yani bir yazıyı) tutabilen bir türdür.

- Int türü işaretli bir tamsayı türüdür. Int türünün uzunluğu sistemden sisteme değişebilir. 32 bit işletim sistemlerinde Int türü 4 byte, 64 bit işletim sistemlerinde Int türü 8 byte uzunluğundadır.

- UInt türü Int türünün işaretsiz biçimidir. Görüldüğü gibi pozitif sınırı Int türünden iki kat daha uzundur.

- Int8 bir byte'lık işaretli tamsayı türüdür. Bunun işaretsiz biçimi UInt8 biçimindedir.

- Int16 iki byte'lık işaretli tamsayı türüdür. Bunun işaretsiz biçimi UInt16 biçimindedir.

- Int32 dört byte'lık işaretli tamsayı türüdür. Bunun işaretsiz biçimi UInt32 biçimindedir.

- Int64 bir byte'lık işaretli tamsayı türüdür. Bunun işaretli biçimi UInt32 biçimindedir.
- Float türü IEEE 754 "short real format"a uygundur. Float türünün yuvarlama hatalarına direnci zayıftır. Bu tür C, C++, Java ve C#'taki float türüyle tamamen aynı formata sahiptir.
- Double türünün yuvarlama hatalarına direnci daha kuvvetlidir. Bu tür de IEEE 754 "long real format"a sahiptir. Dolayısıyla Swift'in Double türü C, C++, Java ve C#'taki double türüyle format bakımından aynıdır.
- Character türü tek bir karakteri tutmak için düşünülmüş bir türdür. Character türü UNICODE karakterleri tutabildiği için 2 byte uzunluktadır.
- Bool türü true, false değerini tutabilen bir türdür.
- String türü birden fazla karakteri tutan (yani bir yazıyı tutan) bir türdür.

Bu kadar çok tür olmasına karşın Swift'te en fazla kullanılan tamsayı türü Int, en fazla kullanılan gerçek sayı türü ise Double türüdür. Biz bir tamsayı türü bildireceksek onu default olarak Int, gerçek sayı türü bildireceksek onu da Double olarak seçmeliyiz. Ancak gerekçelerimiz varsa diğer türleri denemeliyiz.

Swift'in Standart Kütüphanesi

Swift'in de C ve C++'ınki gibi standart bir kütüphanesi vardır. Swift'in standart kütüphanesinde global fonksiyonlar, yapılar, sınıflar, enum'lar ve protokoller bulunmaktadır. Örneğin print fonksiyonu Swift'in bir standart kütüphane fonksiyonudur. Benzer biçimde Array<T> sınıfı da Swift'in bir standart kütüphanesindeki bir sınıftır.

Swift'in standart kütüphanesi yalnızca çok temel fonksiyonları ve sınıfları barındırmaktadır. GUI işlemleri gibi özel işlemleri yapan sınıflar (örneğin Cocoa ya da Cocoa Touch sınıfları) standart kütüphanenin kapsamı dışında bırakılmıştır.

XCode IDE'si

IDE (Integrated Development Environment) yazılım geliştirmeyi kolaylaştıran araçların bir araya getirildiği bir ortamdır. IDE'lerin editörleri ve yardımcı araçları vardır. IDE derleyici değildir. Bir IDE'de biz derleme yapmak istediğimiz zaman IDE de derleyici programını (örneğin swift ya da swiftc) çalıştırarak işlemini yapar. Ancak IDE'ler üretkenlikleri ciddi biçimde arırmaktadır.

Apple'ın geliştirme IDE'sine XCode denilmektedir. XCode eskiden ücretli bir ürün olarak satılıyordu. Daha sonra Apple bunun ücretini 10 doların altına kadar düşürdü, sonra da bedava hale getirdi. XCode IDE'sinin tarihsel gelişimi şöyledir:

- XCode'un ilk versiyonu olan 1.0 2003'te çıktı.
- XCode 2.0 Mac OS X 10.4 Tiger ile birlikte 2005'te çıktı.
- XCode 3.0 Mac OS X 10.5 Leopard ile birlikte 2008'de piyasaya sürüldü.
- XCode 4.0 2010'da çıktı.
- XCode 5.0 2013'te IOS 7 SDK ile birlikte çıktı.

- XCode 6.0 ise 2014 yılında piyasaya çıktı
- XCode 7.0 2015 yılında çıktı.
- XCode'ın 8'li versiyonları 2016'da çıktı.
- XCode'un 9'lu versiyonları da 2017'de çıktı.

Bu ana versiyon numaralarının dışında Apple küçük numaralı sürümler de çıkarmıştır. Kursun yapıldığı sırada XCode'un son versiyonu 9.3'tür.

Genel Sentaks Gösterimi

Programlama dillerinin standartlarında ya da referans kitaplarında dilin sentaksı teknik olarak BNF (Backus-Naur Form) türevi notasyonlarla ifade edilmektedir. BNF notasyonu (ISO bunu Extended BNF ismiyle standardize de etmiştir) ikianlamlılığa (ambiguity) izin vermeyen bir sentakstır. Ancak öğrenilmesi biraz zahmetlidir. Biz kursumuzda açısıl parantez ve köşeli parantez tekniğini kullanacağız. Bu gösterimde açısıl parantez içerisindeki ifadeler yazılması zorunlu olan öğeleri, köşeli parantez içerisindeki ifadeler yazılması isteğe bağlı öğeleri belirtir. Diğer atomlar aynı biçimde aynı konumda bulundurulmak zorundadır. Örneğin C'deki if deyiminin sentaksı bu notasyona göre şöyle ifade edilir:

```
if (<ifade>
    <deyin>
else
    <deyin> ]
```

XCode ile Swift Programının Çalıştırılması

XCode IDE'si kullanılarak bir Swift konsol programı şöyle oluşturulabilir:

1) XCode IDE'si çalıştırılır.

2) File/New/Project menüsü seçilir. Buradan OS X/Application/Command Line Tool seçilir ve next yapılır. Sonra da Projeye (Product'a) isim verilerek ilerlenir.

İfade (Expression) Kavramı

Değişkenlerin, operatörlerin ve sabitlerin her bir kombinasyonuna ifade (expression) denir. Örneğin:

```
a = b
a = b * c
foo()
10
a
```

birer ifadedir. Tek başına bir değişken ve tek başına bir sabit ifade belirtir. Ancak tek başına kullanılan bir operatör ifade belirtmez.

Bildirim İşlemi

Bir değişkenin kullanılmadan önce derleyiciye tanıtılması işlemine bildirim (declaration) denilmektedir. Tıpkı C, C++, Java ve C#'ta olduğu gibi Swift'te de değişkenler kullanılmadan önce bildirilmek zorundadır.

Bildirim işleminin genel biçimi kabaca şöyledir:

```
var <isim listesi> : <tür>  
var <isim> = <ilkdeğer> [, ...]  
var <isim>: <tür> = [ilkdeğer] [, ...]  
let <isim> = <ilkdeğer>, [, ...]  
let <isim>: <tür> = [ilkdeğer] [, ...]
```

Bildirim işlemi var ya da let anahtar sözcüğü ile başlatılır. Bu anahtar sözcükleri bildirilecek değişken ismi ve onun türü izler. Tür belirtilmek zorunda değildir. Bu durumda tür verilen ilkdeğerden hareketle otomatik olarak belirlenir. Yani bildirimde tür belirtilmiyorsa değişkene ilkdeğer verilmek zorundadır. Örneğin:

Örneğin:

```
var a: Int = 10, b = 12.3  
print(a, b)
```

Burada a Int türündendir ve ona 10 ilkdeğeri atanmıştır. b için bir tür belirtilmemiştir. Bu durumda b'nin türü atanan ilkdeğerden hareketle derleyici tarafından belirlenir. 12.3 sabiti double türden olduğu için b double türünden ele alınır. Örneğin:

```
var a = 10, b = "ankara"  
print(a, b)
```

Burada a Int türden b ise String türündendir. Eğer bildirimde tür belirtilmemişse ilkdeğer verilmek zorundadır. Ancak tür belirtilmişse ilkdeğer vermek zorunlu değildir. Örneğin:

```
var a          // error!  
var b: Int32   // geçerli
```

Tabii bu işlemler karışık biçimde yapılabilir. Örneğin:

```
var a = 12.3, b: Int, c = 100      // geçerli
```

Aynı türden birden fazla değişken de aşağıdaki gibi bildirilebilir:

```
var a, b: Int
```

Aynı bildirimde bazı değişkenlerin türleri otomatik olarak belirlenebileceği gibi bazılarının türleri de açık olarak belirtilebilir. Örneğin:

```
let a = 10, b: Double = 20, c = 30
```

Burada a Int, b Double ve c de Int türündendir.

Bildirimde var ile let arasında önemli bir farklılık vardır. var ile değişken bildirildiğinde onun değeri değiştirilebilir. Ancak let ile değişken bildirildiğinde o değişken const olur. Onun değeri bir daha değiştirilemez. Örneğin:

```
var x = 10
let y = 20

print(x, y)

x = 30
y = 40      // error!
```

Pek çok dilde const bir değişken bildirilirken -bir daha değer atanamayacağı gerekçesiyle- ilkdeğer verme zorunludur. Swift'te de eskiden bu zorunluluk vardı. Fakat daha sonra yerel değişkenler için bu zorunluluk kaldırıldı. Artık biz let ile bildirdiğimiz const yerel değişkenlere ilkdeğer vermek zorunda değiliz. Tabii bu durumda bildirimde onun türünü belirtmek zorundayız. İlkdeğer verilmemiş let değişkenlere yalnızca bir kez değer atanabilmektedir. İkinci bir değer atanamamaktadır. Örneğin:

```
let y: Int
y = 10      // geçerli
print(y)
y = 20      // error
```

Swift'in resmi "Language Reference" dokümanına göre let ile daha sonra ilkdeğer verme yalnızca yerel değişkenlere özgüdür. Global let değişkenlere daha sonra ilkdeğer verilemez. Global let değişkenlere ilkdeğer verme işlemi yine bildirim sırasında yapılmak zorundadır. (Ancak Swift derleyicileri global let değişkenlere daha sonra ilkdeğer vermeyi de geçerli kabul ediyor.)

Swift'te let ile const değişken kullanma diğer programlama dillerine çok yaygındır. Öyle ki bu dilde eğer bir değişkene bir değer atayıp onun değerini bir daha değiştirmeyeceksek onu var ile değil let ile bildirmeliyiz. Swift için doğru teknik budur. Pek çok programlama dilinde bu konuya özel bir önem verilmemektedir. Hatta Swift derleyicileri uyarı düzeyleri açıldığında içerisindeki değerlerin bir daha değiştirilmediği var ile bildirilmiş değişkenler için uyarı bile vermektedir. Bu durumda iyi bir teknik için programcı hep let ile bildirim yapmaya çalışmalı, eğer bildirdiği değişkeni daha sonra değiştirecekse ancak o zaman onu var haline getirmelidir.

Eğer bildirimde değişkenin türü belirtilmemişse verilen ilkdeğerden tür tespiti yapılmaktadır. Bu durumda verilen ilkdeğer nokta içeriyorsa tür Double olarak, içermiyorsa Int olarak belirlenir. Örneğin:

```
let x = 10.2, y = 23
print(type(of: x), type(of: y))      // Double Int
```

Burada x Double türünden y ise Int türündendir.

İki tırnakla ilkdeğer verme durumunda tür String olarak belirlenir. Örneğin:

```
let s = "Ankara"
print(type(of: s))      // String
```

Swift'te Character türünden sabitler için tek tırnak kullanılmamaktadır. Character sabitleri de yine çift tırnakla belirtilmektedir. Tabii bu durumda çift tırnağın içerisine tek bir karakter yazılır. Örneğin:

```
var c:Character

c = "a"
print(c)          // a
c = "ar"          // error!
```

Swift'te çift tırnak hem String türünden hem de Character türünden sabit belirtmek için kullanıldığına göre tür belirtmemişsek ve değişkene iki tırnak içerisinde ilkdeğer vermişsek o değişkenin türü ne olacaktır? İşte bu durumda iki tırnak içerisinde tek bir karakter varsa bile tür yine String olarak alınmaktadır. Örneğin:

```
let s = "a"
print(type(of: s)) // String
```

Burada s'i artık Character yapmak için bizim türü açıkça Chracter biçiminde belirtmemiz gerekir.

İleride de görüleceği gibi Swift'te aslında tüm temel türler birer yapı belirtmektedir. Bu yapılarında başlangıç metotları (constructors) tür dönüştürmesi için kullanılabilir. Bu durumda aşağıdaki gibi de Caharacter türünden değişken bildirimi yapabiliriz:

```
let s = Character("a")
print(type(of: s)) // Character
```

true ya da false olabilecek Bool türden ilkdeğerler için değişkenin türü Bool olarak belirlenir. Örneğin:

```
let flag = true
print(type(of: flag)) // Bool
```

String Enterpolasyonu

Stringler içerisinde bir tane \ karakteri sonra buna yapışık bir parantez varsa parantez içerisindeki ifade yazıya dönüştürülüp bu kalının yerine insert edilmektedir. Buna string enterpolasyonu denilmektedir. Yani yazı içerisindeki \(\ifade) kalıbı bir yer tutucudur. Bu yer tutucu yerine ifadenin değeri yerleştirilir. Örneğin:

```
let a = 10, b = 20.2
print("a = \(a), b = \(b)") // a = 10, b = 20.2
```

Örneğin:

```
let a = 10, b = 20
print("a + b = \(a + b)") // a + b = 30
```

Fonksiyonların Bildirimleri

Swift çok paradigmalı bir dildir. Swift'te fonksiyonlar tıpkı C++'ta olduğu gibi bir sınıfın ya da yapının içerisinde bulunabileceği gibi global düzeyde de bulunabilmektedir. Böylece Swift'te istersek biz hiç sınıf kullanmadan yalnızca fonksiyonlarla prosedürel teknikte kod yazabiliriz.

Swift'te fonksiyon bildiriminin genel biçimi şöyledir:

```
func <isim>([parametre bildirimi]) [-> <geri dönüş değerinin türü>]
{
    //...
}
```

Örneğin:

```
func foo()
{
    //...
}
```

Burada foo geri biçiminde aralarına dönüş değeri olmayan parametresiz bir fonksiyondur. Fonksiyonun parametre değişkenleri <isim>: <tür> ',' yerleştirilerek bildirilir.

Örneğin:

```
func bar(a: Int, b: Double) -> Double
{
    //...
}
```

Burada bar fonksiyonun birinci parametresi Int türden ikinci parametresi Double türündendir ve geri dönüş değeri de Double türündendir. Parametre bildiriminde tür bilgisi bulundurulmak zorundadır.

Eğer bildirimde parametre parantezinin içi boş bırakılırsa bu durum fonksiyonun parametreye sahip olmadığı anlamına gelir. Eğer parametre parantezinden sonra -> atomu kullanılmamışsa bu durum da fonksiyonun geri dönüş değerinin olmadığı anlamına gelmektedir. Fonksiyonun tek bir geri dönüş değeri vardır. Ancak birden fazla değer geri döndürmenin çeşitli yolları (örneğin demet ile) bulunmaktadır.

return deyimi hem fonksiyonu sonlandırmak için hem de geri dönüş değerini oluşturmak için kullanılır. return deyiminin genel biçimi şöyledir:

return [ifade]

```
func add(a: Int, b: Int) -> Int
{
    print("I am add")

    return a + b
}

let result = add(a: 10, b: 20)
print(result)
```

Eğer fonksiyonun geri dönüş değeri yoksa return kullanılabilir; ancak yanına bir ifade yazılamaz. Örneğin:

```
func add(a: Int, b: Int)
{
    print(a + b)
```



```

    return
}

add(a: 10, b: 20)

```

Geri dönüş değeri olmayan bir fonksiyonda biz return kullanmak zorunda değiliz. Zaten akış fonksiyonun sonuna geldiğinde fonksiyon sonlanacaktır. Ancak geri dönüş değeri olan bir fonksiyonda return kullanılmak zorundadır. Fonksiyonun her mümkün akışı için return deyiminin görülmesi gerekmektedir. Örneğin:

```

func add(a: Int, b: Int) -> Int    // error!
{
    print("I am add")
}

```

Fonksiyon parametre bildiriminde parametrenin türleri kesinlikle belirtilmek zorundadır. Örneğin:

```

func foo(var a)                // error!
{
    //...
}

```

Bir fonksiyonun geri dönüş değerinin olması onu kullanmayı zorunlu hale getirmez. Ancak geri dönüş değeri olduğu halde programcı çağırım sırasında bundan faydalanmamışsa Swift derleyicisi bu durumda bir uyarı mesajı vermektedir.

Fonksiyonun parametre değişkenleri eskiden let ve var ile bildirilebiliyordu. Swift'in 3 versiyonuyla birlikte parametre bildiriminde let ve var anahtar sözcükleri kaldırıldı. Bunun yerine inout anahtar sözcüğü getirildi. Normal olarak default durumda fonksiyonun parametre değişkenleri const durumdadır yani fonksiyon tarafından bunlar değiştirilemez. Örneğin:

```

func foo(a: Int)
{
    a = 10                // error!
    //...
}

```

Ancak parametre değişkenini değiştirmek istiyorsak artık inout anahtar sözcüğünü kullanmak zorundayız. Örneğin:

```

func foo(a: inout Int)
{
    a = 10                // geçerli
    //...
}

```

inout anahtar sözcüğünün tür belirten anahtar sözcüğün önüne getirildiğinde dikkat ediniz. inout bir parametre değişkenine atama yapıldığında bu çağırımdaki değişkenin değerinin değişmesine yol açmaktadır. Bu konu ileride ele alınacaktır.

Fonksiyonların Çağırılması

Fonksiyon çağırma işleminin genel biçimi şöyledir:

```
<fonksiyon ismi>([argüman listesi])
```

Bir fonksiyon çağrılırken parametre değişkeni sayısı kadar argüman girilmek zorundadır. Fonksiyonların parametre değişkenlerine "parametre (parameter)", fonksiyon çağrılırken girilen ifadeler "argüman (argument)" denilmektedir. Argümanlar aynı türden herhangi bir ifade olabilirler.

Swift'te fonksiyonların parametre değişkenlerinin hem kendi isimleri hem de etiket isimleri vardır. Parametre değişken isimleri fonksiyonunun içerisinde kullanılırken etiket isimleri fonksiyon çağrılırken kullanılmaktadır. Parametre değişkeni bildirilirken etiket isimleri parametre değişken isminin önüne yazılır. Örneğin:

```
func foo(x a: Int, y b: Double)
{
    print("a = \(a), b = \(b)")
}
```

Burada fonksiyonun birinci parametre değişkeninin ismi a, etiket ismi x, ikinci parametre değişkeninin ismi b, etiket ismi de y'dir. Fonksiyonun içerisinde parametre isimlerinin kullanıldığına dikkat ediniz. Örneğin:

```
func foo(width w: Double, height h: Double) -> Double
{
    return w * h
}
```

Fonksiyon çağrılırken argümanlarda etiket isimlerinin belirtilmesi gerekir. Bu belirtme işlemi şöyle yapılmaktadır:

```
[etiket ismi][:]<ifade>
```

Örneğin:

```
func foo(x a: Int, y b: Double)
{
    print("a = \(a), b = \(b)")
}
```

```
foo(x: 10, y: 20)
```

Argümanlarla parametreler aynı sırada girilmek zorundadır. Örneğin:

```
func foo(width w: Double, height h: Double) -> Double
{
    return w * h;
}
```

```
let result = foo(width: 10, height: 200)
print(result)
```

Etiket ismi yerine '_' karakteri kullanılırsa bu durumda çağırma sırasında etiket ismi belirtilmez. Örneğin:

```
func foo(w: Double, _ h: Double) -> Double
{
```

```

    return w * h;
}

let result = foo(w: 10, 200)
print(result)

```

Burada biz ikinci parametrede etiket ismi olarak `_` kullandık. Böylece çağırım sırasında etiket ismi belirtilmemiştir.

Swift'te 3 versiyonundan önce fonksiyonların ve metotların birinci parametrelerinde (fakat yalnızca birinci parametrelerinde) etiket belirtilmemişse bu durum birinci parametrenin etiketsiz olduğu anlamına (yani `_` etiketin sahip gibi anlama) geliyordu. Örneğin:

```

func foo(count: Int, max: Int)
{
    //...
}

```

Başka bir deyişle eskiden yukarıdaki fonksiyonun eşdeğeri şöyleydi:

```

func foo(_ count: Int, max: Int)
{
    //...
}

```

Yani biz bu fonksiyonu eskiden şöyle çağırıyorduk:

```
foo(10, max: 20)
```

Bu kural Swift'in 2'li versiyonlarında sınıfların ve yapıların metotlarında da geçerliydi ancak başlangıç metotlarında (constructor) geçerli değildi. Ancak Swift'in 3 versiyonuyla birlikte bu ilk parametre için olan özel durum kaldırıldı. Yani artık biz ilk parametrede etiket belirtmesek bile parametre ismi aynı zamanda etiket ismi olmaktadır. Başka bir deyişle artık yukarıdaki fonksiyonu biz şöyle çağırmak zorundayız:

```
foo(count: 10, max: 20)
```

Cocoa kütüphanesindeki fonksiyonların ve sınıfların metotlarının parametrik yapıları bu eski sistemdeki gibi olmaya devam etmektedir. Dolayısıyla siz Cocoa'da ilk parametrede etiket kullanılmadığını görürseniz şaşırmayın.

Swift'te Fonksiyon Overload İşlemleri

Bilindiği gibi programlama dillerinde aynı isimli fonksiyon bulunabilme özelliğine "function overloading" ya da "method overloading" denilmektedir. İngilizce'de "overload" "aşırı yükleme" anlamına geldiği için Türkçe kaynakların bir bölümünde böyle ifade edilmektedir.

Swift'te global düzeyde ya da bir sınıf içerisinde aynı isimli fonksiyonlar bulunabilir. Ancak bunun mümkün olabilmesi için aynı isimli fonksiyonların aralarında aşağıdaki farklılıkların birinin ya da birden fazlasının bulunuyor olması gerekir:

- Parametre sayılarının farklı olması

- Parametre türlerinin farklılığı
- Etiket isimlerinin farklılığı
- Geri dönüş değerinin farklılığı

Örneğin:

```
func foo(a: Int)
{
    print("foo1")
}

func foo(b: Int)           // geçerli
{
    print("foo2")
}

foo(a: 100)
foo(b: 200)
```

Bu durum geçerlidir, çünkü fonksiyonların isimleri aynı olsa da onların etiket isimleri farklıdır. (Anımsanacağı gibi eğer biz parametre değişkeninde etiket ismi belirtmezsek parametre isimleri aynı zamanda etiket ismi olmaktadır.)

Örneğin:

```
func foo(x a: Int)
{
    print("foo: \(a)")
}

func foo(y a: Int)           // geçerli!
{
    print("foo: \(a)")
}

foo(x: 10)           // foo: 10
foo(y: 20)           // foo: 20
```

Yukarıdaki iki foo'nun parametre değişkenlerinin etiket isimleri farklıdır. Örneğin:

```
func foo(x a: Int)
{
    print("foo: \(a)")
}

func foo(x a: Double)        // geçerli!
{
    print("foo: \(a)")
}

foo(x: 10)           // foo: 10
foo(x: 20.2)         // foo: 20.2
```

Bu örnekte parametrelerin etiket isimleri aynıdır fakat türleri farklıdır. Örneğin:

```

func foo(x a: Int) -> Int
{
    print("foo: \(a)")

    return 0
}

func foo(x a: Int) -> Double
{
    print("foo: \(a)")

    return 0
}

let x: Int = foo(x: 10)           // foo: 10
let y: Double = foo(x: 10)       // foo: 10

```

Yukarıdaki iki foo'nun da etiket isimleri ve parametre türleri aynıdır. Ancak geri dönüş değerleri farklıdır. İşte derleyici "overload resolution" sırasında bu fonksiyonların geri dönüş değerlerinin hangi türden değişkene atandığına bakarak hangi fonksiyonun çağrıldığını tespit etmeye çalışır. Tabii bunu her zaman başaramayabilir. Örneğin:

```
var result = foo(x: 0)
```

Burada derleyici hangi foo'nun çağrılmış olduğunu anlayamaz. Halbuki:

```
var result: Double = foo(x: 0)
```

burada artık geri dönüş değeri Double olan foo'nun çağrıldığını derleyici anlamaktadır.

Anımsanacağı gibi Swift'te de çağrılan fonksiyonun geri dönüş değeri kullanılmak zorunda değildir. Yani fonksiyonun geri dönüş değerinden biz faydalanabiliriz ya da hiç faydalanamayabiliriz. Bu durumda da sorun oluşabilir. Örneğin:

```
foo(x: 10)
```

Burada da hangi foo'nun çağrıldığı tespit edilememektedir.

Swift'ta aynı isimli bir fonksiyon çağrıldığında hangisinin çağrılmış olduğunun tespiti derleyici için kolaydır. Onların zaten etiketleri farklıysa derleyici etiket farklılığından hangi aynı isimli fonksiyonun çağrıldığını anlayabilmektedir. Eğer etiketler aynı ise fakat parametrelerin türleri farklıysa bu durumda çağrımda kullanılan argümanın türü ile uyuşan aynı isimli fonksiyonun çağrıldığı kabul edilir. Örneğin:

```

func foo(x a: Int)
{
    print("foo: \(a)")
}

func foo(x a: Double)
{
    print("foo: \(a)")
}

```

```

}

foo(x: 100)    // 100 Int türden olduğu için Int parametreye sahip foo çağrılır
foo(x: 100.0) // 100.0 Double türden olduğu için Double parametreye sahip olan çağrılır

```

Swift'te ileride görüleceği gibi farklı türlerin birbirlerine atanması hiçbir durumda geçerli değildir. Örneğin bu dilde Int türü Double türüne doğrudan atanamaktadır. Dolayısıyla derleyicinin aynı isimli hangi fonksiyonun çağrıldığı konusundaki belirleme süreci daha kolaydır. Aşağıdaki durum hakkında ne düşünürsünüz?

```

func foo(x a: Character)
{
    print("foo Character: \(a)")
}

func foo(x a: String)
{
    print("foo String: \(a)")
}

foo(x: "a")    // foo String: a
foo(x: "ankara") // foo String a

```

Swift'ta çift tırnak içerisinde tek karakter her zaman default olarak String biçiminde değerlendirilmektedir. Character için bizim açıkça dönüştürme yapmamız gerekir:

```

foo(x: Character("a"))    // foo Character: a

```

Fonksiyon Parametrelerinin Default Değer Alması Durumu

Default argüman C++'ta öteden beri vardı. C#'a Microsoft Language Specification 3.0 versiyonuyla sokulmuştur. Python'da da bu özellik belli bir süreden beri vardır.

Fonksiyon bildiriminde parametre değişkenine bir değer atanırsa biz o fonksiyonu çağırırken ona karşı gelen argümanı hiç girmeyebiliriz. Bu durumda sanki o argüman için ilkdeğer olarak atanmış değer girilmiş gibi işlem söz konusu olur. Örneğin:

```

func foo(a: Int, b: Int = 100)
{
    print("a = \(a), b = \(b)")
}

foo(a: 10, b: 20)
foo(a: 10)    // foo(10, b: 100)

```

Buradaki foo her zaman iki parametrelidir. Ve her zaman çağırma sırasında iki parametre aktarımı yapılır. Default argüman yalnızca çağırma işlemini kolaylaştırmaktadır. Örneğin:

```

func foo(a: Int = 100, b: Int = 200)
{
    print("a = \(a), b = \(b)")
}

foo(a: 10, b: 20) // foo(10, b: 20)

```

```
foo(a: 10)           // foo(10, b: 200)
foo(b: 20)           // foo(a: 100, b: 20)
foo()                // foo(a: 100, b: 200)
```

Default değer alan parametre değişkeni için argüman girersek artık o default değer bir önemi kalmaz. Örneğin:

```
func printError(message msg: String = "Ok")
{
    print("Error: \(msg)")
}

printError(message: "invalid name")
printError()
```

Default argümana sahip fonksiyonlar çağrılırken argümanların soldan sağa göreli konumları değiştirilemez. Örneğin:

```
func foo(a: Int = 10, b: Int = 20, c: Int = 30)
{
    print("a = \(a), b = \(b), c = \(c)")
}

foo(a: 100, c: 200)    // geçerli!
foo(c: 200, a: 100)    // error!
foo(b: 100, c: 200)    // geçerli
```

Burada çağırma sırasında önce c etiketli sonra a etiketli parametre için argüman giremeyiz.

Default argümanlı fonksiyonlar Swift'in standart kütüphanesinde de karşımıza çıkmaktadır. Örneğin aslında print fonksiyonunun parametrik yapısı şöyledir:

```
func print(_ items: Any..., separator: String = " ", terminator: String = "\n")
```

Burada print fonksiyonunun ilk parametresi değişken sayıda argüman alışı desteklemektedir. Bu konu ileride ele alınacaktır. Yani biz print fonksiyonuna etiketsiz olarak istediğimiz kadar çok argüman girebiliriz. separator parametresi birden fazla argüman girildiğinde onların arasına hangi karakterin getirileceğini belirtir. Bı parametrenin default olarak " " biçiminde tek boşluk aldığını görüyorsunuz. terminator parametresi ise yazma işleminden sonra hangi karakterin ekrana basılacağını belirtir. Bu parametre de default olarak "\n" almıştır. Yani yazdırma işleminden sonra imleç aşağı satırın başına geçer. Bu durumda örneğin:

```
print(a)
```

ile,

```
print(a, separator: " ", terminator: "\n")
```

eşdeğer işleme yol açar. Örneğin:

```
let a = 10, b = 20

print(a, b, separator: ", ")
print("ok")
```

Buradaki çıktı şöyle olacaktır:

```
10, 20  
ok
```

Örneğin:

```
let a = 10, b = 20  
  
print(a, terminator: "")  
print(b)  
print("ok")
```

Çıktı şöyle olacaktır:

```
1020  
ok
```

Default argümana sahip fonksiyonlarla aynı isimli daha az parametreye sahip fonksiyonlar overload edilebilir. Fakat bunların çağırılması sırasında iki anlamlılık (ambiguity) hataları oluşabilir. Örneğin:

```
func foo(a: Int = 100, b: Int = 200)  
{  
    print("a = \(a), b = \(b)")  
}  
  
func foo()  
{  
    print("void")  
}  
  
foo()           // hangi foo?
```

C++ ve C#'ta fonksiyonun bir parametresi default değer almışsa onun sağındakilerin hepsinin default değer alması zorunludur. Swift'te etiket isimleri nedeniyle böyle bir zorunluluk yoktur. Örneğin:

```
func foo(a: Int = 100, b: Int)  
{  
    print("a = \(a), b = \(b)")  
}  
  
foo(b: 200)      // foo(100, 200)
```

İyi bir teknik bakımından parametre değişkenine default değer vermek için o değer çok yaygın kullanılıyor olması gerekir. Aksi takdirde kodu inceleyen kişiler yanlış izlenimlere kapılabilirler. Default değer alan fonksiyon parametrelerine etiket iliştilmemişse yine de default parametreler karışık sırada belirtilebilmektedir. Her ne kadar derleyici bu durumda bir error ya da uyarı mesajı vermiyorsa da böyle bir tasarım çoğu kez anlamsızdır. Örneğin:

```
func foo(_ a: Int = 100, _ b: Int)  
{  
    print("a = \(a), b = \(b)")  
}
```



```
foo(10)           // error!  
foo(10, 20)       // geçerli!
```

İç İçe Fonksiyon Bildirimleri

C türevi dillerde iç içe fonksiyonlar bildirilemez. Her fonksiyon diğerinin dışında bildirilmek zorundadır. Fakat Pascal gibi Python gibi bazı dillerde öteden beri iç içe fonksiyon bildirimi zaten vardır. Swift'te de bir fonksiyonun içerisinde başka bir fonksiyon bildirilebilir. Örneğin:

```
func foo()  
{  
    //...  
    func bar()    // geçerli  
    {  
        //...  
    }  
    //...  
}
```

İçerde bildirilmiş fonksiyonlar ancak içinde bildirildiği fonksiyonlar tarafından çağrılabilirler. Yani yukarıdaki örnekte bar fonksiyonu yalnızca foo fonksiyonu tarafından çağrılabilmektedir. Dış fonksiyonun içerisinde iç fonksiyonun çağrılabilmesi için çağırma işleminin iç fonksiyonun bildiriminden sonra yapılması gerekir. Örneğin:

```
func foo()  
{  
    print("foo")  
  
    bar()    // error!  
    func bar()  
    {  
        print("bar")  
    }  
}
```

Global düzeydeki fonksiyonlar için böyle bir sıralama zorunlu değildir. Yani global fonksiyonlarda üstte bildirilmiş fonksiyon altta bildirilmiş fonksiyonu çağırabilir.

```
func foo()  
{  
    print("foo")  
    bar()    // geçerli  
}  
  
func bar()  
{  
    print("bar")  
}  
  
foo()
```

İç fonksiyon dış fonksiyonun yerel değişkenlerini kullanabilir. Onları değiştirebilir de. Örneğin:

```
func foo()
```

```

{
    var a = 10
    print("foo")

    func bar()
    {
        print("bar")
        print(a)           // 10
        a = 20

    }
    bar()
    print(a)               // 20
}

foo()

```

İç fonksiyon dış fonksiyonun herhangi bir yerinde bildirilebilir. Ancak bildirildiği yer önemlidir. Şöyle ki: İç fonksiyon dış fonksiyonun başından kendi bildirimine kadar olan bölgede (yani kendisinden daha yukarısında) bildirilmiş olan dış fonksiyonun yerel değişkenlerini ve parametre değişkenlerini kullanabilir. Örneğin:

```

func foo()
{
    print("foo")

    func bar()
    {
        print("bar")
        a = 20           // error
    }
    var a = 10
    bar()
    print(a)
}

```

İç içe fonksiyon bildirimleri birden fazla kademe olarak da yapılabilir. Yani fonksiyonun içerisinde bir fonksiyon, onun içerisinde başka bir fonksiyon, onun içerisinde de başka bir fonksiyon bildirilebilir.

Sabitler

Swift'te C, C++, Java ve C'taki gibi sabitlerin türleri yoktur. Sabitler değişkenlere atanırken onlar için uygunluk değerlendirmesi yapılmaktadır. Şöyle ki eğer bir sabit ilgili türün sınırları içerisinde kalıyorsa o türe atanabilir. Tür belirtilmediği durumda daha önceden de bahsedildiği gibi nokta içermeyen tamsayı değerler için değişkenler Int türü olarak, nokta içeren değerler için de Double türü olarak değerlendirilmektedir. İki tırnak içerisindeki yazılar tek karakterli bile olsa tür belirtilmemişse ilgili değişken String türünden kabul edilir. Örneğin:

```

let x = 10           // x Int türünden
let y = 10.2         // y Double türünden
let z = "a"          // String türünden

```

Sabitler Swift'te 5 gruba ayrılmaktadır:

1) Tamsayı Sabitleri (Integer Literals)

- 2) Gerçek Sayı Sabitleri (Floating Point Literals)
- 3) Stringler (String Literals)
- 4) Bool Sabitleri (Boolean Literals)
- 5) Nil Sabiti (Nil Literals)

Noktası olmayan sayılar tamsayı sabitleridir. Tamsayı sabitleri default olarak 10'luk sistemde yazılmış gibi ele alınır. Tamsayı sabitleri Swift'te 2'lik sistemde, 8'lik sistemde ve 16'lık sistemde de belirtilebilirler. Şöyle ki:

- Sayı 0b ile başlatılarak yazılırsa sayının 2'lik sistemde (binary) yazılmış olduğu kabul edilir. Örneğin:

```
let x = 0b1010
print(x)          // 10
```

- Sayı 0o ile başlatılarak yazılırsa sayının 8'lik (octal) sistemde yazıldığı kabul edilir. Örneğin:

```
let x = 0o12345          // 8'lik sistemde. x Int türden
```

- Sayı eğer 0x ile başlatılarak yazılırsa sayının 16'lık sistemde (hexadecimal) yazıldığı kabul edilir.

Swift'te sabit yazarken rakamlar arasına istenildiği kadar '_' karakteri konulabilir. Bu kural gruplama yoluyla okunabilirliği artırmak için düşünülmüştür. Örneğin:

```
let x = 1_000_000
print(x)          // 1000000
```

Sayı nokta içeriyorsa böyle sabitlere gerçek sayı sabitleri denilmektedir. Gerçek sabitleri üstel biçimde ya da 16'lık sistemde belirtilebilirler. 16'lık sistemde belirtme yapılırken üstel kısım p ile 10'luk sistemde belirtme yapılırken e ile gösterilmektedir. p'nin yanındaki üstel sayı 2'nin üzeri olarak 16'lık sistemde e'nin yanındaki sayı ise 10'un üzeri olarak 10'luk sistemde yazılmak zorundadır. Örneğin:

```
1.23
1.23e-4
100e30
-100E30          // -100 * 10 üzeri 30
0x123.3Fp4       // 0x123.3F * 2 üzeri 4
```

String'ler iki tırnak içerisinde yazılan karakterlerden oluşmaktadır. Bool sabit olarak yalnızca true ve false anahtar sözcükleri bulunmaktadır. nil anahtar sözcüğü nil sabit belirtir.

Swift'te sabitlerin türü yoktur. Biz bir sabiti var ya da let bildirimi ile ya da daha sonra bir değişkene atıyorsak sabit o değişkenin sınırları içerisinde kalıyorsa sanki o türdenmiş gibi işlem görür, sorun oluşmaz. Kalmıyorsa error oluşur. Örneğin:

```
let x: Int16
x = 40000          // error!
print(x)
```

Burada x Int16 türündendir. Bu türün limitleri [-32768, +32767] arasındadır. Dolayısıyla 40000 değeri bu türün sınırları dışında kaldığı için error oluşur.

Seçeneksel (Optional) Türler ve nil Sabiti

Swift'te her T türünün T? biçiminde temsil edilen (tür belirten sözcükle '?' karakteri bitişik yazılmak zorundadır) seçeneksel T biçimi vardır. Örneğin Int türünün seçeneksel biçimi Int? biçimindedir. String türünün seçeneksel biçimi String? biçimindedir. T? türünden bir değişken hem T türünün tüm değerlerini tutabilir hem de nil özeli değerini tutabilir. Örneğin:

```
var a: Int?  
  
a = 10      // geçerli  
a = nil     // geçerli
```

Halbuki T türünden bir değişkene biz nil değerini atayamayız. Örneğin:

```
var a: Int  
a = nil      // error!
```

O halde T bir tür belirtmek üzere T? türünden bir değişken hem T türünün tüm değerlerini tutabilir hem de ayrıca nil özel değerini tutabilir.

Peki nil sabitinin anlamı nedir? nil sözcük olarak null ile eşanlamlıdır (yani hiçbirşey, boş, yok gibi anlamlara geliyor). Bir değişkenin içerisinde nil varsa o değişkenin içerisinde geçerli bir değer yok gibi düşünülmelidir. Örneğin:

```
func foo() -> Int?  
{  
    //...  
}
```

Biz bu foo fonksiyonunu çağırdığımızda bundan elde edeceğimiz her Int değeri bizim için anlamlı olabilir. Ancak bu fonksiyon başarısız da olabilir. İşte örneğin fonksiyon başarısız olmuşsa bize nil değerini verebilir. O halde bizim bu fonksiyonu çağırdıktan sonra geri döndürülen değer nil olup olmadığını kontrol etmemiz gerekir. Örneğin bir sayının karekökünü geri dönüş değeri olarak veren mysqrt fonksiyonu şöyle yazılmış olabilir:

```
import Foundation  
  
func mysqrt(a: Double) -> Double?  
{  
    if a < 0 {  
        return nil  
    }  
    return sqrt(a)  
}
```

Burada mysqrt eğer parametrenin değeri negatif ise negatif sayıların karekökleri olmadığı için nil değeri ile geri dönmektedir. Tabii bu durumda fonksiyonu çağırırken bizim nil kontrolü yapmamız gerekir. Örneğin:

Örneğin:

```
import Foundation
```

```

func mysqrt(a: Double) -> Double?
{
    if a < 0 {
        return nil
    }
    return sqrt(a)
}

let x = -12.3
let result = mysqrt(a: x)
if result == nil {
    print("negati sayıların karekökü olmaz!")
}
else {
    print(result!)
}

```

Seçeneksel bir türün içerisindeki değer nil ile == ve != operatörleriyle karşılaştırılabilmektedir.

Swift'te Java ve C#'a benzemeyen biçimde referans türlerine biz nil yerleştiremeyiz. Onlara nil yerleştirebilmemiz için onların da seçeneksel bildirilmiş olması gerekir. Örneğin, Sample bir sınıf belirtiyor olsun:

```

var s: Sample
//...
s = nil           // error!

```

Fakat örneğin:

```

var s: Sample?
//...
s = nil           // geçerli

```

T? türünden T türüne otomatik dönüştürme yoktur, halbuki T türünden T? türüne otomatik dönüştürme vardır. Örneğin biz Int bir değişkeni Int? türünden bir değişkene atayabiliriz ancak Int? türünden bir değişkeni Int türünden bir değişkene atayamayız.

Peki T? türünden bir değişkenin içerisindeki değeri nasıl alırız? Bu işleme Swift terminolojisinde "açma (unwrapping)" denilmektedir. Bunun için tek operandlı son ek ! operatörü kullanılır. Örneğin:

```

let a: Int?
var b: Int

a = 10           // geçerli
// b = a         // error!
b = a!           // geçerli!
print(b)

```

Peki ya T? türünden değişkenin içerisinde nil varsa ne olur? İşte bu durumda program çalışırken "açma (unwrapping)" noktasında "exception" oluşur. Örneğin:

```

let a: Int?
var b: Int

a = nil          // geçerli

```

```
b = a!           // geçerli fakat exception oluşacak!
print(b)
```

O halde ancak biz T? türündeki değişkenin içerisinde nil olmadığından emin olursak onu açmalıyız. Örneğin:

```
let a: Int?
var b: Int

a = 100           // geçerli
//...
if a != nil {
    b = a!
    print(b)
}
```

Bu işlemi bir arada yapmak için if deyimine bir özellik eklenmiştir. Bu özellik if deyiminin anlatıldığı yerde ele alınacaktır.

Ayrıca seçeneksel türlere ilişkin "seçeneksel zincir oluşturma" isminde bir erişim biçimi de vardır. Fakat bu erişim biçimi sınıf ve yapılarla ilgili olduğu için orada ele alınacaktır.

Aslında T? türü Swift'in standart kütüphanesindeki Optional<T> türü ile eşdeğerdir. Yani

```
var x: Int?
```

bildirimi ile aslında,

```
var x: Optional<Int>
```

bildirimi eşdeğerdir. Başka bir deyişle biz T? yerine eşdeğer olarak Optional<T> de kullanabiliriz. Optional<T> türü Swift'in standart kütüphanesinde enum biçiminde bildirilmiştir. enum türü ileride ele alınacaktır.

Swift'in ileri versiyonlarında bir de ? yerine ! ile bildirilen otomatik seçeneksel tür (implicit optional type) kavramı da vardır. Otomatik seçeneksel türler tür isminden sonra ona yapışık olarak ! karakteri ile ifade edilirler. Örneğin:

```
var a: Int!
```

Burada a yine seçeneksel bir türdür. Yani a'nın türü Optional<Int> biçimindedir. Bunun ?'li bildirimden tek farkı otomatik açma (unwrap) yapılmasıdır. Yani burada biz a'yı kullandığımızda a'nın içerisindeki değeri kullanmış oluruz. Tabii eğer a'nın içerisinde nil değeri varsa exception oluşur. Böylece otomatik seçeneksel türler sayesinde seçeneksel türden nesnelerin içerisindeki değerleri almak için sürekli ! operatörü kullanmamıza gerek kalmaz. Yine otomatik seçeneksel türe ilişkin değişkenler nil karşılaştırmalarına sokulabilir. Örneğin:

```
if a == nil {
    //...
}
else {
    //...
}
```

Bu türler de aslında yine if deyiminin let ya da var'lı biçimleriyle işleme sokulabilmektedir.

Swift'te Farklı Türlerin Birbirlerine Atanması ve Farklı Türlerin İşleme Sokulması

Swift'te otomatik dönüştürme kavramı yoktur. Bilindiği gibi Java, C# gibi dillerde bilgi kaybına yol açmayan dönüştürmeler (yani atamalar) otomatik yapılmaktadır. Örneğin C# ve Java'da int türünden bir değer long türünden bir değişkene atanabilir. Bu durumda error oluşmaz. Halbuki Swift'te bu işlem error'e yol açmaktadır. Örneğin:

```
var a: Int8 = 10, b: Int32

b = a          // Java ve C#'ta geçerli, Swift'te error!
```

Eğer biz bunu gerçekten yapmak istiyorsak temel türlerin başlangıç metotları yoluyla (yani nesne yaratarak) yapmalıyız. Bu işlemi bir tür dönüştürmesi (type cast) gibi düşünebilirsiniz. Örneğin:

```
var a: Int8 = 10, b: Int32

b = Int32(a)    // bir çeşit cast gibi düşünülebilir
print(b)
```

Benzer biçimde büyük türden küçük türe de bu biçimde atama yapabiliriz. Örneğin:

```
var a: Int8, b: Int32 = 10

a = Int8(b)
print(a)
```

Gerçek sayı türlerinden tamsayı türlerine bu biçimde dönüştürme yapılırken sayının noktadan sonraki kısmı atılır. Örneğin:

```
var d: Double = 12.9
var i: Int

i = Int(d)
print(i)        // 12
```

Büyük türden küçük türe dönüştürme yapılırken büyük türün belirttiği değer küçük türün sınırları içerisinde kalıyorsa sorun oluşmaz. Ancak kalmıyorsa exception oluşur. Exception programın çalışma zamanına ilişkin bir kavramdır. Exception oluştuğunda eğer ele alınmazsa program çöker. Örneğin:

```
var a: Int = 12
var b: Int8

b = Int8(a)
print(b)        // Sorun yok, 12 elde edilir
```

Örneğin:

```
var a: Int = 12000
var b: Int8

b = Int8(a)
```

```
print(b)           // exception oluşur!
```

Swift'te farklı türleri de işleme sokamayız. Örneğin C# ve Java'da biz bir int değerle bir double değeri toplayabiliriz. Bu dillerde "işlem öncesi otomatik tür dönüştürmesiyle" küçük tür büyük türe dönüştürülür ve sonuç büyük tür türünden çıkar. Halbuki Swift'te böyle bir işlem geçerli değildir. Error'le sonuçlanır. Örneğin:

```
var a: Int = 10, b: Double = 20, c: Double  
c = a + b           // error!
```

Bu işlem Swift'te şöyle yapılmalıdır:

```
var a: Int = 10, b: Double = 20, c: Double  
c = Double(a) + b    // geçerli
```

Ancak Swift'te biz tamsayı sabitleriyle gerçek sayı sabitlerini sabit olarak işleme sokabiliriz. Bu tür işlemlerde nihai ürün Double türünden olur. Örneğin:

```
var a: Double  
a = 10 + 20.3         // geçerli  
print(a)
```

Swift'te atama işlemlerinde eğer atanan değer sabit ifadesi ise bu sabit ifadesinin sayısal değeri derleme aşamasında hesaplanır. Eğer bu değer hedef türün sınırları içerisinde kalmıyorsa derleme zamanında error oluşur. Örneğin:

```
var a: Int8  
a = 300               // error!  
a = 100 + 200         // error!  
print(a)
```

Fakat yine aynı iki türün birbirlerine atanmasında taşmalar oluşabilir. Bu durumda çalışma zamanı sırasında exception oluşur. Örneğin:

```
var a: Int8, b: Int8 = 100  
a = b + 30           // exception oluşur!  
print(a)
```

Özetle Swift'te aynı tür üzerindeki taşmalar eğer derleme aşamasında belirlenebiliyorsa (sabit ifadeleri ile taşmalar derleme aşamasında belirlenebilmektedir) error oluşur. Eğer taşmalar derleme aşamasında belirlenemiyorsa çalışma zamanı sırasında exception oluşur.

Taşmalar yalnızca atama sırasında değil her türlü işlemde exception'a yol açarlar. Örneğin:

```
var a: Int8 = 100, b: Int8 = 100, c: Bool  
c = a + b == 100      // Taşma var exception oluşur  
print(c)
```

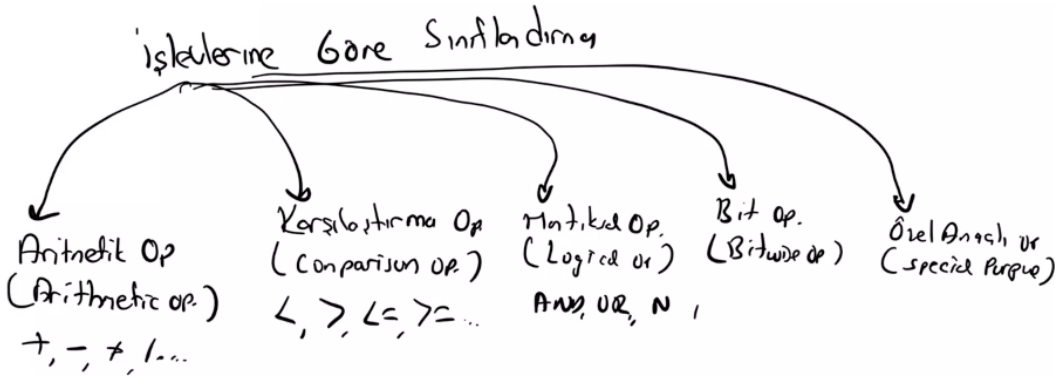

Burada $a + b$ ifadesinde bir taşma oluşacaktır. Çünkü $a + b$ işleminin sonucu `Int8` türündendir. Halbuki bu iki değerin toplamı `Int8`'in sınırları içerisinde değildir.

Swift'te Temel Operatörler

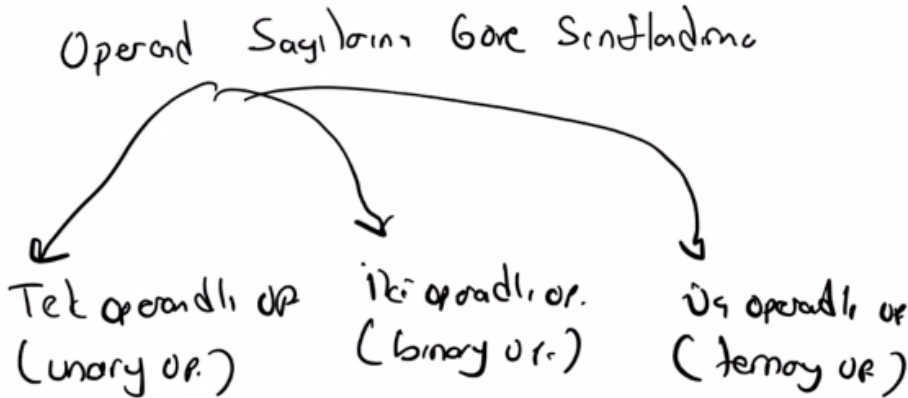
Bir işleme yol açan ve o işlem sonucunda bir değer üretilmesini sağlayan atomlara operatör denir. Swift'teki temel operatörler C, C++, Java ve C# dillerindeki operatörlere çok benzemektedir. Operatörler genellikle üç biçimde sınıflandırılırlar:

- 1) İşlevlerine Göre
- 2) Operand Sayılarına Göre
- 3) Operatörün Konumuna Göre

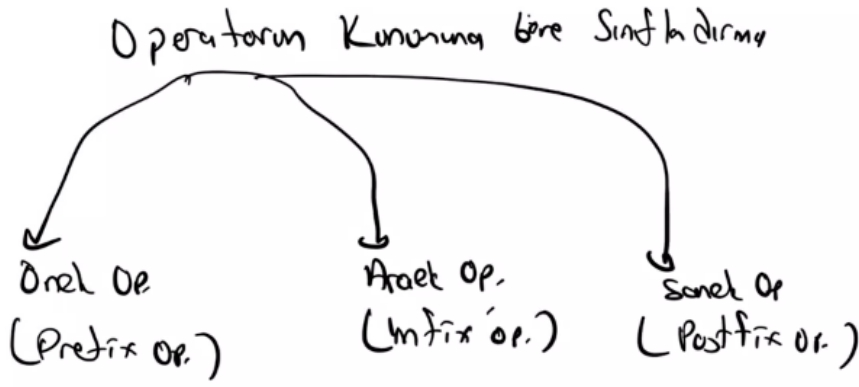
İşlevlerine göre sınıflandırmada operatörler ne çeşit işlem yaptıklarına göre sınıflandırılırlar. Tipik sınıflandırma şöyledir:



Operatörlerin işleme soktuğu ifadelere operand denilmektedir. Örneğin $a + b$ ifadesinde $+$ operatördür, ancak a ve b bu operatörün operandlarıdır. Operand sayılarına göre operatörler üç gruba ayrılırlar:



Operatörün konumuna göre operatörler üç biçimde sınıflandırılırlar. Bazı operatörler operandlarının önüne getirilerek kullanılır. Bazıları sonuna getirilerek kullanılır. Bazıları da arasına getirilerek kullanılmaktadır.



Bir operatörü teknik olarak tanımlayabilmek için bu üç sınıflandırmadaki yerlerini belirtmek gerekir. Örneğin "/" operatörü iki operandlı araek (binary infix) bir aritmetik operatördür" gibi.

Swift'te operatörlerin kullanımı konusunda birkaç önemli ayrıntı vardır:

1) İki operandlı operatörlerin her iki tarafında en az bir boşluk karakteri bulundurulur ya da her iki tarafında boşluk karakteri bulundurulmaz. Örneğin:

```
c = a + b      // geçerli
c = a+b        // geçerli
c = a+ b       // error!
c = a +b       // error!
```

2) Tek operandlı operatörlerin operand'ları bitişik yazılmak zorundadır. Ayrıca tek operandlı önek operatörlerin sol tarafında en az bir boşluk karakterinin, tek operandlı sonek operatörlerin de sağ tarafında en az bir boşluk karakterinin bulundurulması gerekir. Örneğin:

```
b = ! a        // error! tek operandlı operatör ile operandı arasında boşluk yok
c = a &&!b      // error! tek operandlı önek operatörlerin sol tarafında en az bir boşluk karakteri
               // gerekir
c = a && !b     // geçerli
a = !!b        // error!
a = !(!b)      // geçerli
```

Swift'te Operatör Öncelikleri

Bilindiği gibi operatörler paralel işleme sokulmazlar. Belli bir sırada yapılırlar. Her operatörün çıktısı diğerine girdi yapılmaktadır. Örneğin:

```
a = b + c * d
```

```
İ1: c * d
İ2: b + İ1
İ3: a = İ2
```

Derleyici de bu işlemleri yapacak makine komutlarını belli sırada oluşturmaktadır. Swift'teki operatörlerin çoğu zaten diğer yaygın dillerdekilerle aynıdır. Fakat operatör öncelikleri bakımından Swift ile diğer diller arasında bazı önemli farklılıklar vardır. Swift operatörlerinin öncelik tablosu aşağıdaki gibidir:

() [] .	Soldan Sağa
+ - ! ~	Sağdan Sola
<< >>	Yok
/ % * &* &/ &% &	Soldan Sağa
+ - &+ &- ^	Soldan Sağa
..< ...	Yok
is as	Yok
< > <= >= == != === !== ~=	Yok
&&	Soldan Sağa
	Soldan Sağa
??	Sağdan Sola
?:	Sağdan Sola
= *= /= %= += -= <<= >=	Sağdan Sola
&= ^= = &&= =	

Görüldüğü gibi Swift'te bit operatörlerinin önceliği C, C++, Java ve C#'a benzememektedir. Swift'te bu operatörler oldukça önceliklidir. Ayrıca Soldan-Sağa ve Sağdan-Sola (associativity) dışında tabloda bir de "Yok" ibaresi vardır. Yok ibaresi bulunan satırlardaki operatörler birlikte kombine edilemezler. Örneğin:

```
a = b >> 2 << 3 // error!
```

Aritmetik Operatörler

*, /, + ve - iki operandlı aritmetik operatörlerdir. Klasik dört işlemi yaparlar. % operatörü diğer dillerde olduğu gibi "bölümden elde edilen kalan değerini" üretir. Örneğin:

```
let a = 20 % 3
print(a) // 2
```

% operatörü şöyle tıpkı C, C++, Java ve C#'ta olduğu gibi çalışmaktadır.

$$\begin{array}{r} a \overline{) b} \\ \underline{c} \end{array} \quad b \times c + k = a$$

Burada örneğin -20 % 3 işleminden -2 elde edilir. (Python, R gibi bazı dillerde kalan her zaman sağ taraftaki operand ile aynı işaretli olmaktadır. Dolayısıyla bu dillerde -20 % 3 işlemi 1 vermektedir.)

```
let a = 20 % -3
print(a) // 2
```

% operatörünün her iki operandı da tamsayı türlerine ilişkin olmak zorundadır.

Swift'te eskiden ++ ve -- operatörleri vardı. Fakat 3 versiyonu ile birlikte dilden kaldırıldı.

İşaret + ve işaret - operatörleri tek operand'lı örnek aritmetik operatörlerdir. İşaret - operatörü operandının negatif değerini, işaret + operatörü de operandıyla aynı değeri üretir.

Anımsanacağı gibi Swift'te taşma durumları eğer derleme aşamasında tespit edilemiyorsa programın çalışma zamanı sırasında exception'a yol açmaktadır. İşte Swift'te taşma durumunda exception oluşturmayan beş özel aritmetik operatör de vardır. Bunlar &+, &-, &*, &/ ve &% operatörleridir. Örneğin:

```
var a: Int8 = 100, b: Int8
b = a &+ 100      // -56
print(b)
```

Bu operatörler gerçek sonucun yüksek anlamlı byte'larını atarak nihai sonucu elde ederler. Örneğin;

```
var a: UInt8 = 255, b: UInt8
b = a &+ 1      // 0
print(b)
```

Karşılaştırma Operatörleri

Swift'te C, C++, Java ve C#'ta olduğu gibi 6 karşılaştırma operatörü vardır:

< > <= >= == !=

Bu operatörler Bool türden değer üretmektedir. Örneğin:

```
var a: Bool

a = 10 > 20
print(a)
```

Swift'te karşılaştırma operatörleri birbirleriyle kombine edilememektedir. Yani örneğin aşağıdaki gibi bir işlem geçerli değildir:

```
d = a > b < c
```

Swift'te == operatörünün yanı sıra ayrıca === biçiminde özdeşlik operatörü de vardır. Bu operatör referans türlerinde iki referansın aynı nesneyi gösterip göstermediğini sorgulamakta kullanılır.

Mantıksal Operatörler

Swift'te yine üç mantıksal operatör vardır: &&, || ve ! operatörleri. && ve || operatörlerinin diğer dillerde olduğu gibi Swift'te de kısa devre özelliği vardır. Örneğin:

```
func foo() -> Bool
{
    print("foo")

    return true
}

func bar() -> Bool
```

```
{
    print("foo")

    return false
}
```

```
let result = foo() || bar()
print(result)
```

Burada işlemin foo fonksiyonu true ile geri döndüğü için kısa devre özelliğine göre artık hiç bar çağrılmayacaktır. Yukarıdaki örnekte biz &+ yerine normal + operatörünü kullansaydık taşma nedeniyle exception oluşup

?? Operatörü

?? operatörüne İngilizce "nil coalescing operator" denilmektedir. Bu operatör iki operandlı araek bir operatördür. Bu seçeneksel ifadelerde kullanılır. a ?? b ifadesi aşağıdaki ile eşdeğerdir:

```
a != nil ? a! : b
```

Burada a T? türünden b de T türünden olmak zorundadır. a eğer nil ise operatör bize b değerini verir, fakat a nil değilse a'nın içerisindeki değeri verir. Örneğin:

```
var a, b: Int?

a = 10
b = a ?? 100
print(b)          // 10

a = nil
b = a ?? 100
print(b)          // 100
```

Görüldüğü gibi ?? operatörü sol taraftaki operand nil olsa bile bize nil olmayan sağdaki değeri vermektedir.

Koşul Operatörü

Koşul operatörü Java ve C#'taki gibidir. ? atomunun solunda Bool türden bir ifade bulunmak zorundadır. Bu operatör bu ifade true ise ? ile : arasındaki ifadenin değerini false ise : atomunun sağındaki ifadenin değerini üretir. Örneğin:

```
let a = 10
let b = a > 0 ? 100 : 200
print(b)
```

Swift'te Klavyeden Okuma Yapmak

Swift'e readLine fonksiyonu klavyeden ENTER tuşuna basılana kadar bir yazı alır ve onu String? olarak bize verir. Okuma sırasında hiç giriş yapılmadan ENTER tuşuna basılırsa boş string elde edilmektedir. Ancak stdin dosyasının sonuna gelindiye nil değeri elde edilir.) Örneğin:

```
print("Bir yazı giriniz:", terminator:"")
```

```
var str: String? = readLine()
print(str!)
```

İçerisinde yazısal biçimde sayı bulunan bir String'i nümerik türlere dönüştürebilmek için bu türlere ilişkin yapıların String parametrelili init metotları kullanılır. Tabi bu metotlar bize T türünden değil T? türünden bir değer vermektedir. (Çünkü yazının içerisindeki sayı geçersizse nil değeri elde edilir.) Bu durumda örneğin biz klavyeden Int bir değer şöyle okuyabiliriz:

```
print("Bir sayı giriniz:", terminator:"")

var str: String? = readLine()
let number = Int(str!)
print(number!)
```

Örneğin:

```
print("Bir sayı giriniz:", terminator:"")

var str: String? = readLine()
let number = Int(str!) ?? 0
print(number)
```

Tabii bu işlemleri tek aşamada aşağıdaki gibi de yapabiliriz:

```
print("Lütfen bir sayı giriniz:", terminator:"")
var result: Int

result = Int(readLine()!)?
print(result * result)
```

Swift'te Deyimler

Bu bölümde Swift'teki temel deyimler ele alınacaktır. Swift'teki deyimlerde genel olarak anahtar sözcüklerden sonraki parantezler zorunlu değildir. Ancak deyimleri bloklamak (tek deyim olsa bile) zorunlu tutulmuştur. Her ne kadar deyimlerde parantez kullanımı kabul edilse de Swift stiline alışmak için kursumuzda parantezler kullanılmayacaktır.

if Deyimi

if deyiminin genel biçimi şöyledir:

```
if [()] <Bool türden ifade []] {
    //...
}
[
else {
    //...
}
]
```

Görüldüğü gibi if anahtar sözcüğünden sonra parantezler zorunlu değildir. Fakat if deyimindek ifade Bool türden olmak zorundadır. if'in doğruysa ve yanlışsa kısmı bloklanmak zorundadır. if deyiminin yine else kısmı olmak zorunda değildir. Örneğin:

```
print("Lütfen bir sayı giriniz:", terminator:"")
let val: Int = Int(readLine()!!)

if val % 2 == 0 {
    print("girilen değer çift")
}
else {
    print("girilen değer tek")
}
```

else-if için blok açmaya gerek yoktur. else anahtar sözcüğünden sonra hemen if gelebilir. Örneğin:

```
print("Lütfen bir sayı giriniz:", terminator:"")
var val: Int = Int(readLine()!!)

print("Lütfen bir sayı giriniz:", terminator:"")
let val: Int = Int(readLine()!!)

if val == 1 {
    print("bir")
}
else {
    if val == 2 {
        print("iki")
    }
    else {
        if val == 3 {
            print("üç")
        }
        else {
            print("diğer bir sayı")
        }
    }
}
```

Ancak istersek yine blok açabiliriz. Örneğin:

```
print("Lütfen bir sayı giriniz:", terminator:"")
let val: Int = Int(readLine()!!)

if val == 1 {
    print("bir")
}
else {
    if val == 2 {
        print("iki")
    }
    else {
        if val == 3 {
            print("üç")
        }
        else {
```

```

        print("diğer bir sayı")
    }
}

```

Örneğin ikinci derece bir denklemin köklerini bulan bir fonksiyon şöyle yazılabilir:

```

import Foundation

func dispRoots(_ a: Double, _ b: Double, _ c: Double)
{
    let delta = b * b - 4 * a * c

    if delta >= 0 {
        let x1 = (-b + sqrt(delta)) / ( 2 * a)
        let x2 = (-b + sqrt(delta)) / ( 2 * a)
        print("x1 = \(x1), x2 = \(x2)")
    }
    else {
        print("Kök yok!..")
    }
}

var a, b, c: Double

print("a:", terminator: "")
a = Double(readLine()!)

print("b:", terminator: "")
b = Double(readLine()!)

print("c:", terminator: "")
c = Double(readLine()!)

dispRoots(a, b, c)

```

Swift'te if deyiminin bildirimli bir biçimi de vardır. Bu bildirimli biçimde bildirilen değişkene ilkdeğer verilmesi zorunludur. Verilen ilkdeğerin de seçeneksel bir türe ilişkin olması gerekir. Bu durumda bu bildirilen seçeneksel değer nil değilse if deyimi doğrudan sapar, nil ise yanlıştan sapar. Örneğin:

```

print("Bir sayı giriniz:", terminator:"")
let s = readLine()!

if let x = Int(s) {
    print("Değer nil değil: \(x)")
}
else {
    print("değer nil")
}

```

Burada if'teki x değişkenine seçeneksel bir türden ilkdeğer verilmek zorundadır. Örneğin burada ilkdeğer olarak Int fonksiyonu kullanılmıştır. Bu fonksiyonun String overload versiyonu Int? türünden bir değer verir. İşte buradaki if deyimi aslında bu ilkdeğerin nil olup olmadığına bakmaktadır. Ayrıca böyle if deyimlerindeki bildirilen değişken artık seçeneksel değildir. Dolayısıyla yukarıdaki örnekteki x de Int? türünden değil doğrudan Int türündendir. Başka bir deyişle bu if kalıbında verilen ildeğer T? türündense bildirilen değişken T türünden olur.

Bu nedenle biz bu değişkeni yalnızca if deyiminin doğruysa kısmında kullanabiliriz. else kısmında kullanamayız. Yukarıdaki if deyiminin eşdeğeri şöyle oluşturulabilir:

```
print("Bir sayı giriniz:", terminator:"")
let s = readLine()!

let x = Int(s)
if x != nil {
    print("Değer nil değil: \(x)")
}
else {
    print("değer nil")
}
```

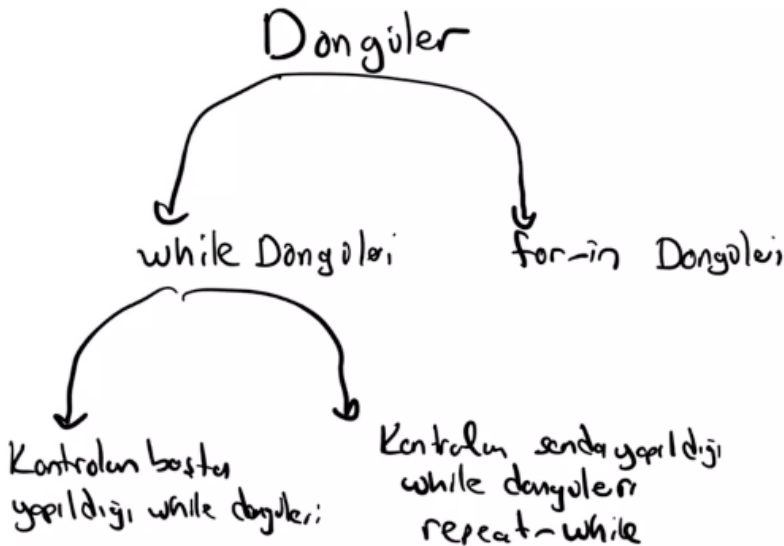
Tabii bu biçimdeki if deyimlerinde değişken let yerine var ile de bildirilebilir. Bu durumda biz o değişkenin değerini değiştirebiliriz. Örneğin:

```
print("Bir sayı giriniz:", terminator:"")
let s = readLine()!

if var x = Int(s) {
    print("Değer nil değil: \(x)")
    x = 0
    print("x'in yeni değeri: \(x)")
}
else {
    print("değer nil")
}
```

Döngü Deyimleri

Swift'te üç döngü deyimi vardır: while döngüleri ve for-in döngüleri. Yine while döngüleri kendi aralarında "kontrolün başta yapıldığı while döngüleri" ve "kontrolün sonda yapıldığı while döngüleri" biçiminde ikiye ayrılmaktadır. Eskiden Swift'te klasik for döngüleri vardı. Swift 3 ile bu döngüler tamamen dilden kaldırılmıştır. Artık for döngüsü olarak yalnızca for-in döngüleri vardır.



Kontrolün başta Yapıldığı while Döngüleri

Swift'te while döngüleri koşul sağlandığı sürece yinelemeye yol açar. Genel biçimi şöyledir:

```
while [()] <Bool türden ifade> [()] {  
    //...  
}
```

while anahtar sözcüğünden sonra Bool bir türden bir ifadenin bulundurulması zorunludur. Döngü bu ifade true olduğu sürece yinelemeye yol açar. Yine genel biçimden görüldüğü gibi ifadedeki parantezler zorunlu değildir. Ancak döngü deyimlerinin bloklanması zorunludur. Örneğin:

```
var i = 0  
  
while i < 10 {  
    print("\(i)", terminator: " ")  
    i += 1  
}  
print()
```

Kontrol Sonda Yapıldığı while Döngüleri (repeat-while)

Kontrolün sonda yapıldığı while döngüleri Swift 2.0'a kadar diğer dillerde olduğu gibi do-while anahtar sözcükleriyle kuruluyordu. Ancak swift 2.0'dan sonra do-while yerine repeat-while anahtar sözcükleri tercih edilmiştir. Ancak Semantik bir değişiklik yoktur. Döngünün genel biçimi şöyledir:

```
repeat {  
    //...  
} while [()] <Bool türden ifade> [()]
```

Örneğin:

```
var i = 0  
  
repeat {  
    print("\(i)", terminator: " ")  
    i += 1  
} while i < 10  
print()
```

for-in Döngüleri

Bu döngüler diğer pek çok dildeki foreach döngüleri gibidir. for-in döngülerinin genel biçimi şöyledir:

```
for <değişken ismi> in <Iterator nesnesi> {  
    //...  
}
```

for-in döngüleri genel olarak "dolaşılabilir (iterable)" nesnelerle kullanılabilir. Dolaşılabilir nesneler teknik anlamda Iterator protokolünü destekleyen sınıflar türünden nesnelerdir. Tipik olarak Range sınıfı, listeler, demetler (tuples), sözlükler dolaşılabilir nesnelerdir. for-in döngüleri şöyle çalışır: Döngünün her yinelenmesinde dolaşılabilir nesnenin sıradaki elemanı döngü değişkenine yerleştirilir ve döngü yinelenir. Ta ki dolaşılabilir

nesnedeki elemanlar bitene kadar. Tamsayılardan dolaşılabilir bir nesne yaratmak için "aralık operatörü (range operator)" kullanılmaktadır. Aralık operatörü ... ile temsil edilmektedir ve iki operandlı aralık bir operatördür. a...b ifadesi sonucunda [a, b] aralığındaki tamsayılar elde edilmektedir. Örneğin:

```
let r = 1...10
print(r)
print(type(of: r))
```

Burada CountableClosedRange<Int> türündendir. Buradaki sınıf Generic bir sınıftır ve genel olarak bir aralık belirtir. Aralık yarı açık da olabilir. Bu durumda ..< operatörü kullanılır. Örneğin:

```
let r = 1..
```

Burada [1, 10) aralığında tamsayılar elde edilecektir. Her iki aralık operatöründe de sol taraftaki operandın sağ taraftaki operand'tan küçük ya da ona eşit olması gerekir. Aksi takdirde çalışma zamanı sırasında exception oluşmaktadır.

Aralık operatörünün operandları gerçek sayı türlerine ilişkin de olabilir. Örneğin:

```
let r = 1.2...3.4
print(type(of: r), r)
```

Burada r ClosedRange<Double> türündendir. Tabii aralık operatöründe operandların aynı türden olması gerekir. Ancak operandlar sabit olarak verilmişse biri noktalı bir değer diğeri noktasız bir değer de olabilir.

Aralık operatörü bize dolaşılabilir bir sınıf nesnesi verdiği için biz onu for-in deyiminde kullanabiliriz. Örneğin:

```
for i in 1...10 {
    print(i, terminator: " ")
}
print()
```

Örneğin asallık testi yapan bir program şöyle yazılabilir:

```
func isPrime(_ val: Int) -> Bool
{
    if val < 2 {
        return false
    }

    for i in 2..
```

```

    }
}
print()

```

Ters sırada dolaşım için aralık operatöründen elde edilen nesne ilgili sınıfın `reversed` metoduyla ters çevrilebilir. Ancak bu durumda aralık operatörünü paranteze almak gerekir. Örneğin:

```

for i in (0..<10).reversed() {
    print(i, terminator: " ")
}
print()

```

`for-in` döngüsünde biz tamsayılardan oluşmayan bir aralık kullanamayız. Örneğin:

```

for r in 1.0...3.0 {    // error!
    print(r)
}

```

`for-in` döngülerinde tılamalı aralık oluşturmak için `stride` fonksiyonu kullanılmaktadır. `stride` fonksiyonunun üç etiketi vardır: `from`, `to` ve `by`. Fonksiyon generic olduğu için değişiklik türlerle çalışabilmektedir. Örneğin:

```

let s = stride(from: 10, to: 1, by: -2)
print(s)

```

`stride` fonksiyonuyla `for-in` döngülerini istediğimiz gibi oluşturabiliriz. Örneğin:

```

for i in stride(from: 10, to: 1, by: -0.5) {
    print(i, terminator: " ")
}
print()

```

Bu sayede örneğin daha önce yazmış olduğumuz `isPrime` fonksiyonunu daha etkin bir biçimde şöyle de yazabiliriz:

```

import Foundation

func isPrime(_ val: Int) -> Bool
{
    if val < 2 {
        return false
    }
    if val % 2 == 0 {
        return val == 2
    }

    for i in stride(from: 3, to: Int(sqrt(Double(val))), by: 2) {
        if val % i == 0 {
            return false
        }
    }
    return true
}

for i in 2...1000 {

```

```

    if isPrime(i) {
        print(i, terminator: " ")
    }
}
print()

```

Biz stride fonksiyonu ile gerçek sayılardan oluşan bir yineleme sağlayabiliriz. Örneğin:

```

for r in stride(from: 1.0, to: 10.0, by: 0.5) {    // geçerli
    print(r)
}

```

for-in döngü değişkeni her zaman let biçimindedir. Biz döngü değişkenine yeni bir değer atayamayız. Örneğin:

```

for i in 1...10 {
    i = i + 1           // error!
    print(i)
}

```

for-in döngülerindeki döngü değişkenlerinin faaliyet alanları yalnızca for-in döngüsü ile sınırlıdır. Bu değişkenler döngü dışında kullanılamazlar. Örneğin:

```

for i in 1...10 {
    print(i)
}
print(i)           // error!

```

Bazen for-in döngülerinde döngü değişkenini hiç kullanmayız. Amacımız n defa bir işi yinelemek olabilir. Bu durumda döngü değişkeni yerine '_' karakteri yerleştirilir. Örneğin:

```

for _ in 1...10 {
    print(".", terminator: "")
}
print()

```

break ve continue Deyimleri

C, C++ Java ve C#'ta olduğu gibi break deyimi döngü deyiminin kendisini sonlandırmak için, continue ise döngünün içerisindeki deyimi sonlandırmak için (yani yeni bir yinelemeye geçmek için kullanılır.) Örneğin:

```

var i = 0

while true {
    print("Bir sayı giriniz:", terminator: "")
    if let val = Int(readLine()!) {
        if val == 0 {
            break
        }
        print(val * val)
    }
}

```

Burada programın akışı break deyimini gördüğünde döngü sonlanır. Örneğin:

```
import Foundation

var val: Double
while true {
    print("Bir sayı giriniz:", terminator: "")
    val = Double(readLine()!)
    if val == 0 {
        break
    }
    if val < 0 {
        print("negatif sayıların gerçek kökleri yoktur!")
        continue
    }
    print(sqrt(val))
}
```

Burada programın akışı break anahtar sözcüğünü gördüğünde döngü sonlanacak, continue anahtar sözcüğünü gördüğünde de döngü yeni bir yinelemeye girecektir.

İç içe döngülerde break ve continue deyimleri default durumda yalnızca kendi döngüsü için işlem yapmaktadır. Ancak Swift'te Java'da olduğu gibi etiketli break ve continue deyimleri de vardır. Böylece iç içe döngülerde break ya da continue kullanırken hangi döngüden çıkılacağı ya da hangi döngü için sonraki yinelemenin yapılacağı belirtilebilir. Etiketler döngü deyimlerinin başına getirilmek zorundadır. Örneğin:

```
var str: String

all:
    for i in 1...10 {
        for k in 1...10 {
            print("(\\(i), \\(k))", terminator: "")
            str = readLine()!
            if str == "q" {
                break all
            }
        }
    }
}
```

Etiketler döngü deyimleriyle aynı satırda belirtilebilir ya da daha yukarıdaki bir satırda da belirtilebilir. Tabii etiketlerden sonra hemen döngü deyiminin gelmesi gerekmektedir. Aynı örnek continue için de şöyle verilebilir:

```
var str: String

all:
    for i in 1...10 {
        for k in 1...10 {
            print("(\\(i), \\(k))", terminator: "")
            str = readLine()!
            if str == "q" {
                continue all
            }
        }
    }
}
```

switch Deyimi

Swift'in switch deyimi C, C++, Java ve C#'inkinden oldukça farklı ve geniştir. Deyimin genel biçimi şöyledir:

```
switch [(ifade listesi)] {  
    case <ifade listesi>:  
        //...  
    case <ifade listesi>:  
        //...  
    case <ifade listesi>:  
        //...  
    ...  
    [default]:  
        //...  
}
```

switch deyimi case bölümlerinden ve default bölümünden oluşmaktadır. Deyim şöyle çalışır: Derleyici switch anahtar sözcüğünün yanındaki ifadenin değerini hesaplar. Sonra yukarıdan aşağıya doğru case bölümlerini inceler. Programın akışı ilk uygun olan case bölümüne aktarılmaktadır. Eğer uygun case bölümü bulunamazsa default bölüm çalıştırılır. default bölüm bulundurulmak zorunda değildir. Örneğin:

```
print("Bir değer giriniz:", terminator: "")  
let val = Int(readLine()!)
```

```
switch val {  
    case 1:  
        print("bir")  
    case 2:  
        print("iki")  
    case 3:  
        print("üç")  
    default:  
        print("diğer")  
}
```

Yukarıdaki örnekte ilk göze çarpan şey muhtemelen case bölümlerinin break'siz sonlandırılmış olmasıdır. Gerçekten de Swift'in switch deyiminde break olmasa da case bölümleri akış diğer case bölümüne geldiğinde otomatik sonlandırılır. (Yani başka bir deyişle case bölümlerinin sonunda sanki break varmış gibi bir etki söz konusu olmaktadır.) Bu nedenle default durumda switch deyiminde "fall through" yoktur. Ancak case bölümlerinde fallthrough anahtar sözcüğü kullanılırsa akış aşağıya doğru düşer. Örneğin:

```
print("Bir değer giriniz:", terminator: "")  
let val = Int(readLine()!)
```

```
switch val {  
    case 1:  
        print("bir")  
        fallthrough  
    case 2:  
        print("iki")  
        fallthrough  
    case 3:  
        print("üç")  
        fallthrough
```

```

    default:
        print("diğer")
}

```

fallthrough anahtar sözcüğünün case bölümlerinin sonuna yerleştirilmesi zorunlu tutulmamıştır. Ancak uygunsuz yerleşimlerde derleyiciler uyarı mesajı verebilirler. Örneğin:

```

case 1:
    ifade1
    fallthrough
    ifade2          // uyarı
case 2:
    //...

```

Tabii bir koşul altında da fallthrough uygulanabilir. Örneğin:

```

case 1:
    ifade1
    if koşul {
        fallthrough
    }
    ifade2
case 2:
    //...

```

Fakat switch deyiminin sonunda bulunan case bölümünde ya da default bölümünde fallthrough kullanılamaz. Yani başka bir deyişle switch deyiminin son bölümü case ise fallthrough kullanamayız, default ise de fallthrough kullanamayız. (Aşağıda daha fazla case ya da default olmadığı gerekçesiyle). Örneğin:

```

switch val {
    case 1:
        print("bir")
        fallthrough
    case 2:
        print("iki")
        fallthrough
    case 3:
        print("üç")
        fallthrough
    default:
        print("diğer")
        fallthrough          // error!
}

```

Her ne kadar Swift'in case bölümleri default olarak zaten break'li ise de biz yine switch içerisinde break deyimini kullanabiliriz. Örneğin:

```

switch val {
    case 1:
        print("bir")
        break
    case 2:
        print("iki")
        break
    case 3:

```



```

        print("üç")
        break
    default:
        print("diğer")
        break
}

```

Tabii bu örnekte aslında break deyimini kullanmanın bir anlamı yoktur. Fakat case bölümünü erken bitirmek gerekebilir. Örneğin:

```

case 1:
    if koşul {
        break
    }
    ifade1
    ifade2

```

switch deyiminde bir case bölümlerine en az bir deyim yerleştirilmek zorundadır. Örneğin:

```

case 1:                // error!
case 2:
    print("iki")

```

Burada case 1 durumu için bir deyim bulundurulmamıştır. Tabii fallthrough da bir deyim statüsünde olduğu için buraya fallthrouh yerleştirebiliriz:

```

case 1: fallthrough
case 2:
    print("iki")

```

Belli bir case bölümünde ya da default bölümde hiçbir şey yapma istemiyorsak o zaman break kullanmalıyız. Çünkü Swift'te bir boş deyim yoktur. Örneğin:

```

switch val {
    case 1:
        print("bir")
    case 2:
        print("iki")
    case 3:
        print("üç")
    default:
        break
}

```

Swift'in switch deyiminde tüm olasılıklar case içerisinde değerlendirilmek zorundadır. (Swift'in resmi dokümanlarında bu durum "exhaustive" sözcüğüyle ifade ediliyor). Yani başka bir deyişle switch ifadesinin her değeri için blok içerisinde çalıştırılacak bir kodun bulunması gerekir. Bu durum default bölümün kullanılma olasılığını artırmaktadır. Örneğin:

```

var val: Int
print("Bir sayı giriniz:", terminator:"")
val = Int(readLine()!)

switch val {                // error: switch must be exhaustive, consider adding a default clause

```

```

    case 1:
        print("bir")
    case 2:
        print("iki")
    case 3:
        print("üç")
}

```

Yukarıdaki switch deyimi derleme sırasında error oluşturacaktır. Çünkü val değişkeninin alacağı değerlerin hepsi işlenmiş durumda değildir. Deyime eklenecek default bölümü geri kalan tüm değerlerin işleneceğini belirttiğinden error'ü ortadan kaldıracaktır.

Swift'in switch deyiminde default bölüm sonda bulunmak zorundadır. (Halbuki C, C++, Java ve C#'ta default bölüm herhangi bir yerde bulunabilir.)

Swift'te switch deyiminin case bölümleri sabit ifadesi olmak zorunda değildir. (Bu durumda aynı değere sahip case bölümlerinin olup olmadığının derleme aşamasında denetlenemeyeceğine dikkat ediniz.) Derleyici case bölümlerine sırasıyla bakar, ilk uygun case bölümünü çalıştırır. Örneğin:

```

var a, b, c: Int

print("a:", terminator:"")
a = Int(readLine()!)

print("b:", terminator:"")
b = Int(readLine()!)

print("c:", terminator:"")
c = Int(readLine()!)

switch c {
    case a:
        print("birinci case")
    case b:
        print("ikinci case")
    default:
        print("hiçbiri")
}

```

Swift'in switch deyiminin case bölümlerinde birden fazla değer bulundurulabilir. Bu durumda değerler ',' atomu ile birbirinden ayrılmalıdır. Örneğin:

```

var val: Int
print("Bir sayı giriniz:", terminator:"")
val = Int(readLine()!)

switch val {
    case 1, 2, 3:
        print("bir ya da iki ya da üç")
    case 4, 5, 6:
        print("dört ya da beş ya da altı ya da yedi")
    default:
        print("hiçbiri")
}

```

case bölümleri aralık da içerebilir. Örneğin:

```
var val: Int
print("Bir sayı giriniz:", terminator:"")
val = Int(readLine()!!)

switch val {
    case 1...3:
        print("bir ya da iki ya da üç")
    case 4...6:
        print("dört ya da beş ya da altı")
    default:
        print("hiçbiri")
}
```

Örneğin:

```
var val: Int
print("Bir sayı giriniz:", terminator:"")
val = Int(readLine()!!)

switch val {
    case 1...3, 7, 8:
        print("bir ya da iki ya da üç ya da yedi ya da sekiz")
    case 4...6, 9...10:
        print("dört ya da beş ya da altı ya da dokuz ya da on")
    default:
        print("hiçbiri")
}
```

Swift'in switch deyiminde switch anahtar sözcüğünün yanındaki kontrol ifadesi gerçek sayı türlerine ilişkin olabilir, String türüne ilişkin olabilir case ifadeleri de gerçek sayı türlerinden ya da String türünden olabilir. Örneğin:

```
var name: String
print("Bir isim giriniz:", terminator: "")

name = readLine()!

switch name {
    case "Ali":
        print("Ali seçildi")
    case "Veli":
        print("Veli seçildi")
    case "Selami":
        print("Selami seçildi")
    default:
        print("Bilinmeyen bir kişi seçildi")
}
```

Switch deyiminde istenirse case bölümlerine bir değişken bildirimi yerleştirilebilir. Bu durum özellikle demetlerde (tuples) kullanılıyor olsa da diğer durumlarda da kullanılabilir. Böyle bir case bölümü her türlü switch değerini alabilecek durumdadır. Bu değer bildirilen değişkene yerleştirilerek kullanılabilir. Örneğin:

```
var name: String
```

```

print("Bir isim giriniz:", terminator: "")

name = readLine()!

switch name {
    case "Ali":
        print("Ali seçildi")
    case "Veli":
        print("Veli seçildi")
    case "Selami":
        print("Selami seçildi")
    case let str:
        print("Başka bir şey girildi: \(str)")
}

```

Tabii böyle bir case bölümünün sonunda bulunması ve yalnızca bir tane olması anlamlıdır. Ayrıca bildirim let yerine var anahtar sözcüğü ile de yapılabilir. Bu durumda bildirilen değişkene atama da yapılabilir. Bu biçimde bildirilen değişkenlerin faaliyet alanları kendi case bölümlerini kapsamaktadır. case bölümlerinde let ya da var ile bildirilen değişkenlerin türleri switch ifadesindeki tür ile aynıdır.

Swift'in switch deyiminde case bölümlerinde değişken bildirimini where koşul cümlesi de izleyebilir. Bu durumda belli koşulları sağlayan switch değerleri belli case bölümleriyle elde edilebilmektedir. Örneğin:

```

var val: Double
print("Bir sayı giriniz:", terminator: "")
val = Double(readLine()!)

switch val {
    case let x where x > 10 && x < 20:
        print("x 10 ile 20 arasında")
    case let x where x >= 20 && x < 30:
        print("x 20 ile 30 arasında")
    default:
        print("hiçbiri")
}

```

Bir switch deyiminde case bölümlerinin hepsinin switch ifadesindeki tür ile aynı türden olması zorunludur. Örneğin:

```

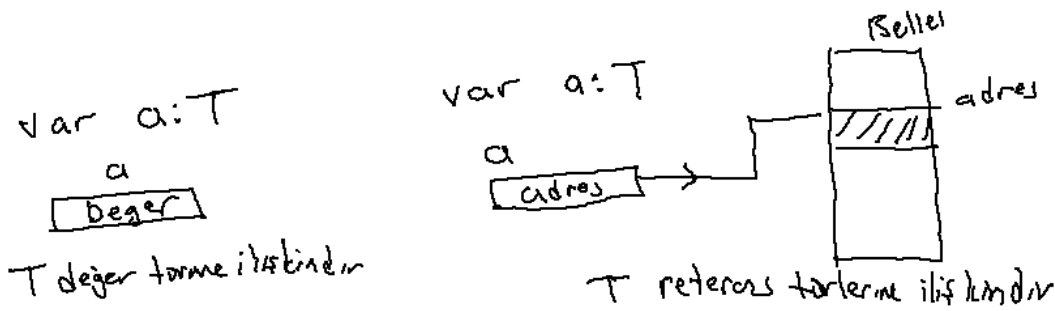
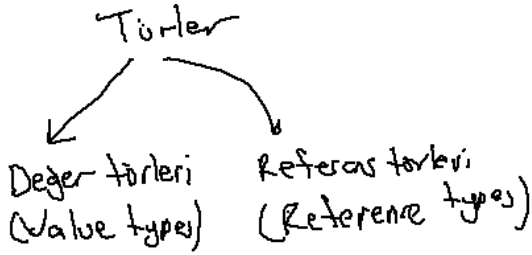
var val: Int
print("Bir sayı giriniz:", terminator: "")
val = Int(readLine()!)

switch val {
    case 1:
        print("Bir")
    case "Ali": // error!
        print("Ali")
    default:
        break
}

```

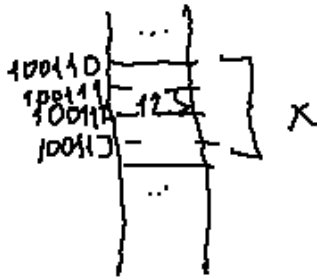
Değer Türleri ve Referans Türleri

Tıpkı C#'ta olduğu gibi Swift'te de türler kategori olarak iki kısma ayrılmaktadır: Değer türleri (value types) ve referans türleri (reference types). T bir tür olmak üzere T türünden bir değişken eğer değerin doğrudan kendisini tutuyorsa T türü kategori olarak değer türlerine ilişkindir. Eğer T türünden bir değişken değeri değil de değerin bulunduğu bellek adresini tutuyorsa T türü kategori olarak referans türlerine ilişkindir. Fiziksel bellekte her bir byte'ın ilk byte sıfır olmak üzere artan sırada bir adres değeri vardır.



Bellekteki her bir byte'a ilk byte sıfır olmak üzere artan sırada bir adres karşı düşürülmüştür. Dolayısıyla her bir değişkenin bellekte bir adresi vardır. Bir byte'tan uzun olan değişkenlerin ve nesnelerin adresleri onların yalnızca en düşük adres değeriyle ifade edilir. Örneğin:

`var x: Int32 = 123`



Burada x'in adresi 100110'dur.

Java, C# ve Swift'teki referans türleri aslında C ve C++'taki gösterici (pointer) türleri gibidir. Swift'te gösterici kavramı yoktur. Ancak referans türleri gösterici gibi davranmaktadır.

Swift "değer türleri ve referans türleri" ayrımını C#'tan almıştır. Swift'te struct ve enum türleri kategori olarak değer türlerine ilişkindir. Fakat sınıflar ve protokoller referans türlerine ilişkindir. Şimdiye kadar görmüş

olduğumuz Int, Double, String türlerinin hepsi aslında birer yapıdır. Dolayısıyla bunlar değer türlerine ilişkindir. Yani örneğin biz Int türden bir değişken bildirdiğimizde o değişken doğrudan değeri tutar. Diğer dillerin aksine Swift'te String ve Array türleri de birer struct biçiminde bildirilmiştir ve bunlar kategori olarak değer türlerine ilişkindir.

Stringler Swift'te -ele alınacağı gibi- C# ve Java'da olduğu gibi değiştirilemez bir değildir. Yani biz Swift'te yaratılmış olan bir string'in karakterlerini istersek değiştirebiliriz. String türü de değer türlerine ilişkin olduğuna göre atama sırasında bir string nesnesinin içeriisindeki yazı tümünden diğer string nesnesine kopyalanmaktadır. Örneğin:

```
var s = "abc", k: String

k = s
print("s = \(s), k = \(k)")    // s = "abc, k = "abc"
s.append("d")
print("s = \(s), k = \(k)")    // s = "abcd, k = "abc"
```

Burada String türü kategori olarak değer türlerine ilişkin olduğu için atama sırasında bir adres ataması değil içerik ataması yapılmıştır. Dolayısıyla atamadan sonra artık s ve k nesneleri farklı iki nesnedir. Birisi değiştirildiğinde diğeri değiştirilmez. O halde biz Swift'teki String türü ile Java ve C#'taki String türünü şöyle karşılaştırabiliriz:

Swift	Java / C#
String nesnesinin karakterleri değiştirilebilir	String nesnesinin karakterleri değiştirilemez
String değer türlerine ilişkindir. String türünden değişken doğrudan yazının kendisini tutar	String referans türlerine ilişkindir. String türünden değişken yazının adresini tutar.
İki String türünden değişkeni birbirine atadığımızda yazılar atanmaktadır.	İki String türünden değişkeni birbirine atadığımızda adresler atanmaktadır. Dolayısıyla iki değişken aynı yazıyı gösterir duruma gelmektedir.

Swift'teki String türünü C++'taki string türüne benzetebiliriz.

Fonksiyonlar ve Metotlar

Swift'te hiçbir sınıfın ya da yapının içerisinde bulunmayan yani global düzeyde bulunan alt programlara fonksiyon (function), bir sınıfın ya da yapının içerisindeki alt programlara da metot (method) denilmektedir. Fonksiyonlar isim belirtilerek parantezlerle çağrılmaktadır. Örneğin:

```
foo(...)
```

Ancak metotlar ilgili sınıf ya da yapı türünden nesnelerle (değişkenlerle) nokta operatörü kullanılarak çağrılır. Örneğin a T türünden bir yapı ya da sınıf nesnesi olsun. Bu yapının ya da sınıfın foo isimli metodu şöyle çağrılmaktadır:

```
a.foo()
```

Swift'te Değişkenlerin Faaliyet Alanları (Scope)

Faaliyet alanı bir değişkenin niteliksiz olarak (yani nokta operatörü olmadan tek başına) kullanılabildiği program aralığına denilmektedir. Swift'te de pek çok dilde olduğu gibi değişkenler dört çeşit faaliyet alanı vardır. Faaliyet alanlarına göre değişkenlere de isimler verilmiştir.

1) Global Değişkenler (Global Variables): Bildirimleri sınıfın, yapının ve fonksiyonların dışında yapılmış olan değişkenlere global değişken denir. Global değişkenler her yerden kullanılabilirler. Örneğin:

```
var g: Int = 10    // global değişken

func foo()
{
    print(g)        // 10
    g = 20
}

foo()
print(g)            // 20
```

Normal olarak bir global değişken yukarıdan aşağıya doğru kullanılmadan önce bildirilmek zorundadır. Ancak global fonksiyonlar önce bildirilip sonra çağrılmak zorunda değildir. Bunların bildirimleri herhangi bir yerde yapılabilmektedir.

2) Yerel Değişkenler (Local Variable): Bir fonksiyon ya da metodun içerisinde bildirilmiş olan değişkenlere yerel değişkenler denir. Yerel değişkenler bildirim yerinden itibaren, bildirildikleri fonksiyon ya da metodun sonuna kadarki bölgede kullanılabilirler. Başka fonksiyonlarda ya da metotlarda kullanılamazlar. Örneğin:

```
func foo()
{
    print("foo")
    a = 20    // error!

    var a: Int

    a = 10
    print(a)    // geçerli
}

func bar()
{
    a = 30    // error!
}
```

Swift'te fonksiyonların ya da metotların içerisinde ana bloktan başka bloklar oluşturulamaz. Halbuki C, C++, Java ve C#'ta bu yapılabilmektedir. Swift de yine aynı metot ya da fonksiyon içerisinde aynı isimli başka bir değişken bildirilememektedir. Aynı isimli global ve yerel değişkenler söz konusu olabilir. Bu durumda fonksiyon ya da metot içerisinde bu isim kullanıldığında yerel olan anlaşılır. Örneğin:

```
var x: Int = 10

func foo()
{
```

```

let x: Int = 20

print(x)      // yerel x
}

print(x)      // global x
foo()

```

3) Fonksiyonların ya da Metotların Parametre Değişkenleri (Parameters): Fonksiyonların ya da metotların parametre değişkenleri yine yerel değişkenler gibi yalnızca o fonksiyon ya da metodun içerisinde kullanılabilmektedir. Tabii fonksiyon ya da metot içerisinde parametre değişkeni ile aynı isimli başka bir değişken bildirilemez. Örneğin:

```

func foo(a: Int)
{
    var a: Int      // error!
    //...
}

```

4) Sınıfların ya da Yapıların Özellikleri (Properties): Bir değişken metotların dışında fakat sınıf ya da yapıların içerisinde bildirilmişse bu değişkenlere "sınıfların ya da yapıların özellikleri (properties)" denilmektedir. Bu tür değişkenler dışarıdan o sınıf ya da yapı türünden nesne ile nokta operatör kullanılarak ya da o sınıfın metotları tarafından doğrudan kullanılabirler. Ancak başka bir yerden kullanılamazlar.

Değişkenlerin Ömürleri Stack ve Heap Kavramları

Bir değişkenin bellekte yer kapladığı zaman aralığına ömür denilmektedir. Şüphesiz bir değişkenin en uzun ömrü çalışma zamanı (runtime) kadar olabilir. Yerel değişkenler programın akışı onların bildirildikleri yere geldiğinde yaratılırlar, akış onların bildirildikleri bloktan çıktığında yok edilirler. Örneğin:

```

func foo()
{
    var a = 10      // değişken yaratıldı
    // ...
    var b = 20      // değişken yaratıldı
    // ...
}                  // akış metottan çıktığında a ve b yok edilir

```

Bir fonksiyon ya da metot çağrılmamışsa onun içerisinde yerel değişkenler henüz bellekte yer kaplamamaktadır. Parametre değişkenleri de fonksiyon ya da metot çağrıldığında yaratılıp akış fonksiyon ya da metottan çıktığında otomatik yok edilmektedir. Tabii fonksiyon ya da metot her çağrıldığında onun yerel değişkenleri yeniden yaratılıp yok edilmektedir. Bu durumda binlerce fonksiyonun ya da metodun bulunduğu durumda bunlar çağrılmadıktan sonra bunların içerisindeki yerel değişkenler bellekte yer kaplamazlar. Yerel değişkenler ve parametre değişkenleri belleğin "stack" diye isimlendirilen bir bölgesinde yaratılıp yok edilmektedir. Stack prosesin bellek alanı içerisinde ve prosese özgüdür. (Aslında her thread'in ayrı bir stack'i vardır.)

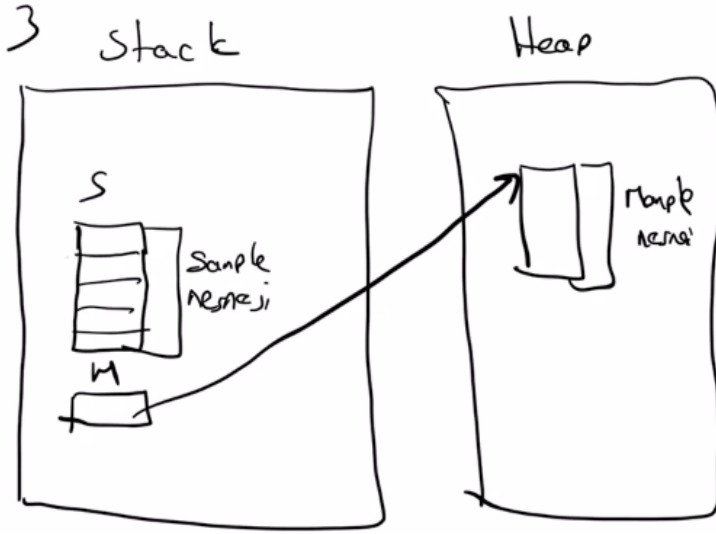
Global değişkenler sürekli bellekte kalırlar. Yani bunların ömürleri programın çalışma zamanı kadardır. Global değişkenler belleğin data ve bss denilen bölümünde tutulurlar. Bir de prosesin adres alanı içerisinde "heap" olarak isimlendirilen bir alan vardır. İşte Swift'te sınıf türünden nesneler (yapı türünden nesneler değil) bunlar "heap" bölgesinde yaratılırlar ve bunların yok edilmesi çöp toplayıcı sistem (garbage collector) tarafından yapılmaktadır.

O halde prosesin adres alanı kabaca dört bölüme ayrılmaktadır: Kod, Data/BSS, Stack ve Heap. Tüm fonksiyonların ve metotların kodları Kod bölümündedir. Örneğin global fonksiyonların kodları, yapıların ve sınıfların içerisindeki metotların kodları hep kod bölümündedir. Global değişkenler Data/BSS bölümünde bulunur. Bunlar program çalıştığı sürece burada yer kaplamaya devam ederler. Stack ise yerel değişkenlerin ve parametre değişkenlerinin yaratılıp yok edildiği deposal doldur boşalt alanıdır. Aslında küçük bir sstack ile binlerce yerel değişken farklı zamanlarda bu alanda yaratılabilmektedir. Heap alanı Swift'te sınıf nesneleri tarafından kullanılmaktadır. Swift'te yapılar heap'te yaratılmazlar. Bunlar global ise Data/BSS alanında yerel ise stack'te yaratılırlar. Halbuki sınıf nesneleri heap'te yaratılmaktadır. Sınıf türünden değişkenler aslında heap'teki sınıf nesneleri gösteren birer referanstır (ya da göstericidir). Örneğin Sample bir yapı, Mample bir sınıf sınıf olsun:

```
func foo()
{
    var s = Sample();
    var m = Mample();
    //...
}
```

Burada s ve m birer yerel değişkendir. Stack'te yaratılmaktadır. s bileşik bir yapı nesnesi olduğu için parçalara sahiptir. Mample nesnesi bir sınıf türünden olduğu için heap'te tahsis edilir. Ancak onun adresi stack'teki m değişkenin de tutulmaktadır.

```
func foo()
{
    var s = Sample()
    var m = Manple()
    /..
}
```



Görüldüğü gibi s tamamı stack'te yaratılan bir yapı nesnesidir. Halbuki m yalnızca adres tutan bir referanstır. Gerçek sınıf nesnesi heap'te yaratılmıştır. Stack'teki tüm nesneler akış fonksiyon ya da metottan çıktığında otomatik yok edilirler. Ancak heap'teki nesneler çöp toplayıcı sistem tarafından artık onları hiçbir referans göstermediği zaman yok edilmektedir.

Nesnelerin stack'te yaratılıp yok edilmesi çok hızlı bir biçimde yapılır. Ancak heap'teki yaratım ve yok edim görelili olarak oldukça yavaştır.

Fonksiyonların inout Parametreleri

Swift'te bir nesneyi adres yoluyla da fonksiyonlara aktarmak mümkündür. Örneğin biz fonksiyon içerisinde yaptığımız değişikliğin asıl diziyi etkilemesini isteyebiliriz (sort işlemi yapan bir fonksiyon düşününüz). Bunun için inout belirleyicisi kullanılır. inout belirleyicisinde argümanın önüne de adres alma anlamına gelen & operatörü getirilmelidir. Örneğin:

```
func foo(_ a: inout Int)
{
    a = 20 // buradaki a aslında çağrılma ifadesindeki argüman olan x
}

var x: Int = 10

print(x) // 10
foo(&x)
print(x) // 20
```

inout parametrelili bir fonksiyon aynı türden bir değişkenle çağrılmak zorundadır. Ayrıca argümanın & operatörü ile niteliklendirilmesi gerekir. inout parametresinin C ve C++'taki göstericiye C#'taki ref parametresine karşılık geldiğine dikkat ediniz. Cocoa kütüphanesinde bazı fonksiyonlar ve metotlar adresiyle aldıkları değişkenlere değer aktarabilmektedir. Dolayısıyla bu fonksiyonlar ve metotlar Swift'te ilgili parametreye karşılık gelen argümanın & operatörü ile niteliklendirilmesiyle çağrılırlar. Örneğin:

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

var a = 10, b = 20
swap(&a, &b)
print("a = \(a), b = \(b)")
```

inout parametresi ile bildirilmiş olan bir fonksiyon çağrılırken ona karşı gelen argümana değer atanmış olması zorunludur. Fonksiyonun bu inout parametreye değer ataması ise zorunlu değildir. inout belirleyici ile var ve let belirleyicileri bir arada kullanılamamaktadır. Ayrıca let bir değişken inout parametresi yoluyla fonksiyona ya da metoda aktarılamaz.

Swift'te Diziler

Elemanları aynı türden olan ve bellekte ardışıl bir biçimde bulunan veri yapılarına dizi (array) denir. Diziyi dizi yapan iki temel özellik vardır:

- 1) Dizi elemanlarının hepsi aynı türdendir.
- 2) Dizi elemanları bellekte ardışıl biçimde tutulur. Böylece dizilerde elemana erişmek çok hızlı (sabit zamanlı) yapılabilmektedir.

Swift'te T bir tür belirtmek üzere dizi türleri [T] biçiminde temsil edilmektedir. Örneğin:

```
var a: [Int]
```

a burada [Int] türündendir yani Int türden bir dizidir.

Swift'te [T] türü tamamen standart kütüphanedeki Array<T> türü ile aynı anlamdadır. Yani:

```
var a: [Int]
```

bildirimi ile:

```
var a: Array<Int>
```

bildirimi tamamen eşdeğerdir. (Array yapısının generic bir yapı olduğuna dikkat ediniz. Generic konusu son bölümlerde ele alınmaktadır.) Bir dizi türünden değişken bildirilmesi o dizinin yaratıldığı anlamına gelmemektedir. Diziler (genel olarak bütün yapılar) ileride görüleceği gibi belli bir sentaksla yaratılmaktadır.

Bir dizi iki biçimde yaratılır:

1) Array<T> yapısının başlangıç metodu yoluyla. Bu yöntemde dizi ilgili yapının başlangıç metoduyla aşağıdaki genel biçimde olduğu gibi yaratılır:

```
<T>(<[argüman listesi]>)
```

ya da

```
Array<T>([arüman listesi])
```

Bu yöntemde dizi doğrudan Array<T> yapısının başlangıç metotları yoluyla yaratılmaktadır. Örneğin:

```
var a: [Int] = [Int]()
```

Burada Int türden içi boş bir dizi yaratılmıştır. Bu bildirimin eşdeğeri şöyledir:

```
var a: [Int] = Array<Int>()
```

Tabii aşağıdaki bildirim de yukarıdakiyle eşdeğerdir:

```
var a: Array<Int> = Array<Int>()
```

Biz dizileri ilkdeğer vererek yaratmak zorunda değiliz. Önce dizi türünden değişken bildirip yaratımı daha sonra da yapabiliriz. Örneğin:

```
var a: Array<Int>
```

```
a = [Int]()
```

Belli bir elemandan belli miktarda olacak biçimde de dizi yaratılabilir. Örneğin:

```
var a:[Int] = [Int](repeating: 0, count: 10)
print(a)
```

Burada a dizisi 10 elemanlı olarak yaratılacaktır ve dizinin her elemanında 0 bulunacaktır. Aynı yaratımı şöyle de yapabiliriz:

```
var a:[Int] = Array<Int>(repeating: 0, count: 10)
print(a)
```

Array<T> yapısının SequenceType protokol parametrelili başlangıç metodu ile de dizi yaratılabilir. Örneğin:

```
var a:[Int] = [Int](1...10)
print(a)
```

Array<T> türü de SequenceType protokolünü desteklediği için mevcut bir dizinin elemanları ile de yeni bir dizi yaratılabilir. Örneğin:

```
var a:[Int] = [Int](1...10)
```

```
var b:[Int] = [Int](a)
```

```
print(a)
print(b)
```

2) Köşeli parantez sentaksıyla: Dizi nesneleri köşeli parantezler içerisine ilkdeğer listesi yazılarak da yaratılabilirler. Örneğin:

```
var a:[Int]

a = [1, 2, 3]
```

Burada dizi Int türden olduğu için köşeli parantezler içerisinde verilen ilkdeğerlerin de Int türden olması zorunludur.

Aslında tür belirtmeden de dizi nesneleri yaratılabilir. Örneğin:

```
var a = [1, 2, 3]
```

Burada verilen ilkdeğerler Int olarak değerlendirildiği için dizi de Int türdendir. Fakat örneğin:

```
var a = [1, 2.2, 3]
```

Burada a dizisi Double türdendir.

Köşeli parantez yoluyla boş bir dizi de yaratılabilir. Örneğin:

```
var a:[Int] = []
```

Tabii bu yaratımı şöyle yapamazdık:

```
var a = [] // error
```

Dizi elemanlarına diğer dillerin çoğunda olduğu gibi [] operatörü ile erişilmektedir. Dizinin ilk elemanı sıfırıncı indekstedir. Bu durumda n elemanlı bir dizinin sonuncu elemanı da n - 1'inci indekste olacaktır. Aslında Swift'te tıpkı C#'ta olduğu gibi bir sınıf türünden referansın ya da bir yapı değişkeninin köşeli parantez operatörü ile kullanılabilmesi için ilgili sınıf ya da yapıda subscript isimli bir elemanın olması gerekir. Subscript C#'taki indeksleyiciler gibi, C++'taki de [] operatör fonksiyonu gibidir. Biz de istersek kendi sınıflarımız ya da yapılarımız için bu subscript elemanlarını yazabiliriz. İşte Array<T> yapısının da elemana erişmekte kullanılan Int parametrelili ve Range<Int> parametrelili subscript'leri vardır. Range<Int> parametrelili subscript sayesinde biz dizinin belli elemanlarını alabiliriz ya da değiştirebiliriz. Örneğin:

```
var a = [1, 2, 3, 4, 5]

print(a[1])
a[0] = 100
print(a)
```

Biz bir aralık vererek dizinin belli elemanlarını da yine bir dizi olarak elde edebiliriz. Örneğin:

```
var a = [1, 2, 3, 4, 5]
var b = a[1...3]
```

```
print(b)
```

Bir aralıkla dizi elemanlarına erişildiğinde elde edilen ürün tek bir eleman olsa bile aynı türden bir dizidir. Bu biçimde elde edilen dizi aslında türü `ArraySlice<T>` türündendir. Benzer biçimde biz aralık belirterek bir grup elemana yine dizi sentaksıyla atama yapabiliriz. Örneğin:

```
var a = [1, 2, 3, 4, 5]

a[1...3] = [10, 20, 30]
print(a)
```

Dizi elemanlarına erişim Java ve C#'ta olduğu gibi programın çalışma zamanı sırasında denetlenmektedir. Dizinin olmayan bir elemanına erişilmek istendiğinde exception oluşmaktadır.

Dizi bildirimi `let` ile yapılırsa dizi elemanları da read-only (başka bir deyişle immutable) olur. Bu durumda biz dizi elemanlarını değiştiremeyiz. Örneğin:

```
let a = [1, 2, 3]

a[0] = 10           // error! dizi read-only
```

Başka bir deyişle biz `let` bir dizinin elemanlarını kullanabiliriz. Ancak herhangi bir elemanını değiştiremeyiz.

Dizilerin uzunlukları da tıpkı Java ve C#'ta olduğu gibi dizi nesnesinin içerisinde saklanmaktadır. Dizi uzunluğu `Array<T>` yapısının `count` propert'si yoluyla elde edilebilir. Propert'ler veri elemanları gibi kullanılan metotlardır. Property konusu ileride ele alınacaktır. Örneğin:

```
var a = [1, 2, 3, 4, 5]
print(a.count)           // 5
```

Diziler `SequenceType` protokolünü desteklediği için `for-in` döngüleriyle kullanılabilir. Yani biz `for-in` döngülerinde her yinelemede dizinin sıradaki bir elemanını elde ederiz. Örneğin:

```
var a = [1, 2, 3, 4, 5]

for i in a {
    print(i)
}
```

Örneğin:

```
let names = ["ali", "veli", "selami", "ayşe", "fatma"]

for name in names {
    print(name)
}
```

Sınıf Çalışması: İçerisinde isimlerin bulunduğu diziye `for-in` ile dolaşarak isimleri aralarında ',' ve bir boşluk getirerek yan yana yazdırınız. Örneğin:

ali, veli, selami, ayşe, fatma

Çözüm:

```
let names = ["ali", "veli", "selami", "ayşe", "fatma"]
var flag = false

for name in names {
    if flag {
        print(terminator: ", ")
    }
    else {
        flag = true
    }
    print(name, terminator: "")
}
print()
```

Şöyle de yapılabilirdi:

```
let names = ["ali", "veli", "selami", "ayşe", "fatma"]

for i in 0..
```

Örneğin:

```
func getMax(array: [Int]) -> Int
{
    var max = array[0]

    for i in 1..
```

Swift'in dizileri zaten büyütülebilen bir yapıdır. Büyütme işlemi kapasite artırımı ile yapılır. Kapasite dizi elemanları için tahsis edilmiş olan uzunluktur. Bu uzunluk capacity property'si ile elde edilebilir. count dizideki dolu olan eleman sayısıdır. Diziye eleman eklendiğinde count bir artar. count değeri capacity değerine geldiğinde tahsis edilmiş alan artırılmaktadır. Yani Array<T> yapısı kendisi içerisinde daha büyük bir alanı tahsis edilmiş

olarak tutmak ister. Böylece yeniden tahsisat (reallocation) işleminin daha etkin yapılmasını sağlar. Kapasite artırımının Swift'in dokümanlarında geometrik olarak (yani eskisinin belirli katı) yapıldığı belirtilmiştir. (Bu tür uygulamalarda genellikle diziler eski uzunluğun iki katı olacak biçimde büyütülmektedir. Array<T> yapısının capacity property'si C#'ın aksine read-only'dir. Yani bunun değerine değiştiremeyiz. Ancak yapının reserveCapacity metodu bu işi yapmaktadır.

Array<T> yapısının append metodu dizinin sonuna eleman eklemekte kullanılır. Metodun parametrik yapısı şöyledir:

```
append(_: Element)
```

Örneğin:

```
var a = [1, 2, 3]

print("count = \(a.count), capacity = \(a.capacity)")
a.append(100)
print("count = \(a.count), capacity = \(a.capacity)")
a.append(200)
print("count = \(a.count), capacity = \(a.capacity)")
print(a)
```

capacity değerinin geometrik olarak dikkat ediniz. Örneğin:

```
var a = [Int]()

for i in 0..<100 {
    print("count = \(a.count), capacity = \(a.capacity)")
    a.append(i)
}
```

Yapının overload edilmiş contentsOf etiketli parametresi bizden bir SequenceType alıp onun içerisindekileri diziye eklemektedir. Yani biz bir grup elemanı tek hamlede böyle ekleyebiliriz. Örneğin:

```
var a = [1, 2, 3]

a.append(contentsOf: [10, 20, 30])
print(a)
```

Yapının insert metodu eklenecek eleman belli bir indekste olacak biçimde eklemeyi yapar. Parametrik yapısı şöyledir:

```
insert(_: Element, at: Int)
```

Örneğin:

```
var a: [Int] = [1, 2, 3, 4, 5]

a.insert(100, at: 2)
print(a)           // [1, 2, 100, 3, 4, 5]
```

Tipki append metodunda olduğu gibi insert metodunun da SequenceType parametrelili overload biçimi de vardır:


```
var a: [Int] = [1, 2, 3, 4, 5]

a.insert(contentsOf: [10, 20, 30], at: 2)
print(a)           // [1, 2, 10, 20, 30, 3, 4, 5]
```

Yapının remove isimli metodu belli bir indeksteki elemanı siler. Metodun parametrik yapısı şöyledir:

```
remove(at: Int) -> Element
```

Metod diziden atılan elemanı bize verir. Örneğin:

```
var a: [Int] = [1, 2, 3, 4, 5]

a.remove(at: 1)
print(a)       [1, 3, 4, 5]
```

Yapının removeFirst ve removeLast metotları parametresizdir. İlk ve son elemanları silmek için kullanılır.

Yapının removeAll metodu tüm elemanları silmek için kullanılır:

```
removeAll(keepCapacity keepCapacity: Bool = default)
```

Örneğin:

```
var a: [Int] = [1, 2, 3, 4, 5, 6, 7, 8]

print("\(a.count), \(a.capacity)")
a.removeAll(keepingCapacity: false)
print("\(a.count), \(a.capacity)")
```

Yapının reserveCapacity metodu kapasiteyi belli bir değere çekmek için kullanılır. Bu metot biz zaten diziye belli miktarda eleman ekleyeceksek boşuna yeniden tahsisat yapılmasın diye kullanılabilmektedir. Metodun parametrik yapısı şöyledir:

```
reserveCapacity(_y: Int)
```

Örneğin:

```
var a = [Int]()

a.reserveCapacity(500)
for i in 0..<500 {
    a.append(i)
}
print("count = \(a.count), capacity = \(a.capacity)")
```

Aslında bu metodun parametresi capacity'nin kesin değerini belirtmez, minimum değerini belirtir. Yani metot kapasiteyi bizim verdiğimiz değerden daha yükseğe de çekebilir.

Yapının contains isimli metodu belli bir elemanın dizide olup olmadığı bilgisini bize Bool olarak verir. Parametrik yapısı şöyledir:

```
contains(_:Element)
```

Örneğin:

```
var a = [23, 45, 21, 16, 9]

print(a.contains(21))
```

index isimli metod belli bir elemanı arar, onun ilk bulunduğu yerin indeks numarasıyla geri döner. Metodun parametrik yapısı şöyledir:

```
index(of: Element) -> Int?
```

Geri dönüş değerinin seçeneksel türden olduğuna dikkat ediniz. Eğer eleman bulunamazsa nil değeri elde edilmektedir. Örneğin:

```
var a = [23, 45, 21, 16, 9]

if let i = a.index(of: 21) {
    print(i)
}
else {
    print("Bulunamdı")
}
```

Array<T> yapısının diğer elemanları Swift'in orijinal dokümanlarından incelenebilir.

İki dizi nesnesi + operatörüyle toplanabilir. Bu toplama işlemi sonucunda yeni bir dizi nesnesi elde yaratılır. Yeni yaratılan dizi nesnesinin içerisinde bu dizilerin elemanları önce soldaki dizinin sonra sağdaki dizinin olacak biçimde bulunur. İşlem sonucunda yeni bir dizi nesnesi elde edilmektedir. Örneğin:

```
let a = [1, 2, 3, 4, 5]
let b = [10, 20, 30, 40, 50]
let c = a + b
print(c)
```

Tabii toplanacak dizilerin aynı türden olması gerekir. Örneğin aşağıdaki toplama işlemi geçerli değildir.

```
let a = [1, 2.5, 3, 4, 5]
let b = [10, 20, 40, 50]
let c = a + b // error!
print(c)
```

Tabii biz bu biçimde birden fazla diziyi de toplayabiliriz. Benzer biçimde diziler üzerinde += işlemi de yapılabilir. Bu durum bir ekleme gibi de düşünülebilir. Ancak ne olursa olsun += işleminde mevcut diziyi ekleme yapılamamakta yeni bir yaratılmaktadır. Örneğin:

```
var a = [1, 2, 3, 4, 5]
let b = [10, 20, 30, 40, 50]
a += b // eşdeğeri: a = a + b
print(a)
```

Array<T> bir yapı olduğu için değer türlerine ilişkindir. Yani Array<T> türünden bir değişken dizi elemanlarının kendisini tutar, dizi nesnesinin adresini tutmaz. Böylece aynı türden iki dizi değişkenini birbirine atadığımızda bir kopyalama söz konusu olacaktır. Atama işleminden sonra birinde yapılan değişiklikler diğerini etkilemeyecektir. Örneğin:

```
var a, b: [Int]

a = [1, 2, 3, 4, 5]
b = a

a[0] = 100
a[1] = 200

print(a)    // [100, 200, 3, 4, 5]
print(b)    // [1, 2, 3, 4, 5]
```

Swift'te böylece dizilerin fonksiyonlara parametre yoluyla geçirilmesi de hep kopyalama yoluyla (call by value) yapılmaktadır. Zaten artık Swift 3 ile birlikte fonksiyonun parametre değişkenleri değiştirilememektedir. Dolayısıyla fonksiyonun diziyi değiştirmesi de inout parametresi kullanmadan mümkün olmaz. Örneğin:

```
var a: [Int] = [1, 2, 3, 4, 5]

func foo(b:[Int])
{
    print(b)

    b[0] = 100    // error!
    b[1] = 200    // error!
}

foo(a)          // [1, 2, 3, 4, 5]
```

Peki dizilerin bu biçimde fonksiyonlara aktarılması performans kaybına yol açmaz mı? İşte eğer biz dizide değişiklik yapmıyorsak (yani dizi inout ile aktarılmamışsa) aslında derleyici arka planda bir optimizasyonla diziyi zaten adres yoluyla fonksiyona aktarmaktadır.

Swift'te çok boyutlu dizi kavramı yoktur (tabii en azından şimdilik). Çok boyutlu diziler adeta dizi dizileri biçiminde organize edilmektedir. Örneğin [[Int]] türü Int dizileri tutan dizi türüdür. Bu tür Array<Array<Int>> ile eşdeğerdir. Örneğin:

```
let a:[[Int]] = [[1, 2, 3], [4, 5], [6, 7, 8]]

for x in a {
    for y in x {
        print(y, terminator: " ")
    }
    print()
}
```

Tabii aslında bildirimde türü hiç belirtmeyebilirdik. Swift derleyicisi verilen ilkdeğerlerden hareketle türün [[Int]] olduğunu anlayabilmektedir. Örneğin:

```
let a = [[1, 2, 3], [4, 5], [6, 7, 8]]
```

Diziler aynı türden değerleri tutabilirler. Dolayısıyla nasıl [Int] bir dizi Int elemanları tutabiliyorsa [[Int]] bir dizi de [Int] elemanları tutabilir. Örneğin:

```
let a: [[Int]] = [[1.5, 2, 3], [4, 5], [6, 7, 8]] // error!
```

Burada artık dizi dizisinin elemanlarının hepsi [Int] türden değildir. Ancak biz bu ilkdeğer verme işleminde dizinin türünü belirtmezsek tek bir Double değerden hareketle derleyici bunun bir [[Double]] dizi olduğuna hükmeder:

```
let a = [[1.5, 2, 3], [4, 5], [6, 7, 8]] // geçerli  
print(type(of: a)) // Array<Array<Double>>
```

Dizi dizilerinde belli bir elemana iki köşeli parantez ile yani a[i][k] biçiminde erişebiliriz. Burada eleman olan dizilerin aynı uzunlukta olma zorunluluğunun bulunmadığına dikkat ediniz. Zaten Java'da da çok boyutlu diziler ancak bu biçimde oluşturulmaktadır. C#'ta buna dizi dizisi (jagged array) denir. C# ayrıca çok boyutlu dizileri de desteklemektedir. C++'ta da hem çok boyutlu diziler hem de dizi dizileri oluşturmak mümkündür. Örneğin:

```
let a = [[1, 2, 3], [4, 5], [6, 7, 8]] // geçerli  
for i in 0..    for k in 0..        print(a[i][k], terminator: " ")  
    }  
    print()  
}
```

Kümeler (Sets)

Kümeler aynı türden fakat farklı elemanları barındırmak için kullanılan veri yapılarıdır. Bir küme (set) aynı elemandan birden fazla kez tutamaz. Küme içerisindeki elemanların tek olması (unique) garanti altına alınmıştır. Kümelere Swift'te Set<T> isimli generic yapıyla temsil edilmektedir. Bir küme tipik olarak bu yapının başlangıç metodu yoluyla aşağıdaki gibi oluşturulabilir:

```
var s: Set<Int> = Set<Int>()
```

Bir küme bir dizi, aralık gibi nesnelerle de oluşturulabilir. Genel olarak SequenceType protokolü destekleyen nesnelerle biz kümeleri oluşturabiliriz. Bu durumda küme o nesnenin elemanlarından oluşturulmaktadır. Örneğin:

```
let s: Set<Int> = Set<Int>([1, 2, 3, 4, 2, 2, 2, 1, 5])  
print(s)
```

Swift'te köşeli parantezler default durumda diziler için kullanılmaktadır. Ancak ilkdeğer verilirken ya da atama yapılırken eğer hedef tür küme ise köşeli parantezlerle küme elemanları belirtilebilir. Örneğin:

```
var s: Set<Int> = [1, 2, 3, 4, 5] // geçerli  
print(s)  
s = [6, 7, 8, 9, 10] // geçerli  
print(s)
```

Aşağıdaki örnekte s bir küme değildir dizidir:

```
var s = [1, 2, 3, 4, 5]           // geçerli
print(type(of: s))               // Array<Int>
```

Aslında ilkdeğer veriliyorsa generic türün yalnızca ismini de kullanabiliriz. Örneğin:

```
let s: Set = [1, 2, 3, 4, 5]      // geçerli
print(s)

let a: Array = [1, 2, 3, 4]       // geçerli
print(a)
```

Set<T> türünün count property'si yine kümenin eleman sayısını verir. isEmpty property'si Bool türündendir. Kümenin boş olup olmadığı bilgisini bize verir. Örneğin:

```
let s: Set<Int> = Set<Int>([1, 2, 3, 4, 2, 2, 2, 1, 5])
print(s)           // [2, 4, 5, 3, 1]
print(s.count)     // 5
print(s.isEmpty)   // false
```

Kümeler dolaşılabilir nesnelerdir. Yani biz bir kümeyi for-in döngüsüyle dolaşabiliriz. Ancak bu dolaşımın bizim kümeye girdiğimiz sırada yapılmasının bir garantisi yoktur. Kümeler genel olarak sıralı (ordered) bir veri yapısı değildir. Örneğin:

```
let s: Set = [1, 2, 3, 4, 5]

for i in s {
    print(i, terminator: " ")
}
print()
```

Set yapısının insert(·) metodu yeni bir elemanı kümeye eklemek için kullanılır. Metot bize işlemin başarısı hakkında bir demet(tuple) vermektedir. Örneğin:

```
var s: Set = [1, 2, 3, 4, 5]
print(s)           // [5, 2, 3, 1, 4]
let t = s.insert(10)
print(s)           // [5, 10, 2, 3, 1, 4]
print(t)           // (inserted: true, memberAfterInsert: 10)
```

Yapının update(with:) metodu belli bir elemanı kümeye eklemek için kullanılır. Ancak eleman olsa bile yeni eleman eskisinin üzerine bindirilir. Örneğin:

```
var s: Set = [1, 2, 3, 4, 5]
print(s)           // [5, 2, 3, 1, 4]
let t = s.update(with: 10)
print(s)           // [5, 10, 2, 3, 1, 4]
print(t)
```

remove(·) metodu belli bir elemanı argüman olarak alır onu bulursa siler. Geri dönüş değeri T? biçimindedir. Eğer eleman varsa silinen elemanı yoksa nil değeri elde edilmektedir. Örneğin:

```

var s: Set = [1, 2, 3, 4, 5]
print(s)      // [5, 2, 3, 1, 4]

if let result = s.remove(3) {
    print(result)
}
else {
    print("eleman yokmuş!")
}

```

Kümeler üzerinde kesişim, birleşim, fark gibi klasik işlemler yapılabilmektedir. `intersection(_)` isimli metot parametre olarak bir küme alır. İki kümenin kesişiminden elde edilen bir kümeye geri döner. Örneğin:

```

let s: Set = [1, 2, 3, 4, 5]
let k: Set = [3, 4, 10, 20, 30]

let result = s.intersection(k)
print(result)

```

`union(_)` metodu ise birleşim işlemini yapar. Örneğin:

```

let s: Set = [1, 2, 3, 4, 5]
let k: Set = [3, 4, 10, 20, 30]

let result = s.union(k)
print(result)

```

`subtracting` metodu fark işlemi yapar. Örneğin:

```

let s: Set = [1, 2, 3, 4, 5]
let k: Set = [3, 4, 10, 20, 30]

let result = s.subtracting(k)
print(result)

```

İki küme `==` ve `!=` operatörleriyle kullanılabilir. Bu durumda onların tamamen aynı elemanlara sahip olup olmadığı test edilmektedir. Örneğin:

```

let s: Set = [1, 2, 3, 4, 5]
let k: Set = [5, 4, 3, 2, 1]

print(s == k)      // true
print(s != k)      // false

```

`isSubset(of:)` ve `isSuperset(of:)` metotları bir kümenin diğer bir kümenin alt kümesi ve üst kümesi olup olmadığını kontrol etmektedir. (Her küme kendisinin alt kümesi ve üst kemesidir.) Örneğin:

```

let s: Set = [1, 2, 3, 4, 5]
let k: Set = [2, 5, 4]

print(s.isSuperset(of: k))    // true
print(k.isSubset(of: s))     // true

```

Benzer biçimde `isStrictSubset(_)` ve `isStrictSuperset(_)` metotları da özalt küme ve öz üst küme testlerini yapmaktadır.

Bir elemanın küme içerisinde olup olmadığını tespit etmek için `contains(_)` metodu kullanılmaktadır. Örneğin:

```
let s: Set = [1, 2, 3, 4, 5]
print(s.contains(1))           // true
```

Yapının `index(of:)` metodu bir elemanın indeksini özel bir tür olarak verir. Buradan elde edilen değer başka metotlara argüman olarak girilebilmektedir. Örneğin:

```
var s: Set = [1, 2, 3, 4, 5]
let index = s.index(of: 5)
s.remove(at: index!)
print(s)
```

Biz burada `index` metodu ile elemanın indeksini `Index?` biçiminde seçeneksel ayrı bir tür olarak elde ettik. `remove(at:)` metodu bizden bu özel türden indeks değeri alıp silme yapmaktadır.

Aslında `Set<T>` türü köşeli parantez ile eleman elde etmemize izin vermektedir. Ancak köşeli parantezlerin içerisine `int` bir indeks değeri değil `Index` türünden özel değeri geçmek gerekir. Örneğin:

```
var s: Set = [1, 2, 3, 4, 5]
let index = s.index(of: 5)
print(s[index!])
```

Yani elemanın yeri tamsayısal bir indeks değeri ile değil `Index` isimli bir tür değeri ile ifade edilmektedir.

Yapının `sorted(_)` isimli metodu küme içerisindeki elemanları sıraya dizerek bize bir dizi biçiminde verir. Örneğin:

```
var s: Set = [10, 2, 4, 7, 8]
let result = s.sorted()
print(result)
```

Set yapısının diğer elemanları Swift dokümanlarından incelenebilir.

Sözlükler (Dictionaries)

Sözlük anahtar-değer çiftlerinden oluşan bilgileri tutan ve anahtar verildiğinde onun değerine hızlı bir biçimde (algoritmik olarak) erişmeyi sağlayan veri yapılarıdır. Swift'te nasıl diziler temel bir veri türü ise benzer biçimde sözlükler de temel bir veri türüdür. Bu veri yapıları programlama dillerinin kütüphanelerinde genellikle hash tabloları ya da dengelenmiş ikili ağaçlar biçiminde oluşturulmaktadır.

Sözlükler Swift'te `Dictionary<K, V>` isimli generic bir yapıyla temsil edilmiştir. Boş bir sözlük nesnesi `Dictionary` yapısının başlangıç metoduyla yaratılabilir. Örneğin:

```
var d: Dictionary<String, Int> = Dictionary<String, Int>()
```

Aynı işlem şöyle de yapılabilirdi:

```
var d = Dictionary<String, Int>()
```

Sözlük türleri [K:V] biçiminde ifade edilebilmektedir. Yani Dictionary<K, V> ile [K: V] aynı anlamdadır. Örneğin biz boş bir sözlüğü şöyle de yaratabiliriz:

```
var d: [String: Int] = [String: Int]()
```

Benzer biçimde aynı şeyi şöyle de yapabiliriz:

```
var d = [String: Int]()
```

Bir sözlük nesnesi köşeli parantezler içerisinde anahtar ve değerler ':' atomu ile ayrılarak da otomatik biçimde yaratılabilir. Örneğin:

```
var d: [String: Int] = ["Eskişehir": 26, "İstanbul": 34, "Kocaeli": 41, "Konya": 42]  
  
print(d)
```

Aynı bildirim tür belirtmeden de aşağıdaki gibi yapılabilirdi:

```
var a: ["Eskişehir": 26, "İstanbul": 34, "Kocaeli": 41, "Konya": 42]
```

Burada yine a değişkeni [String: Int] türündendir. Benzer biçimde aşağıdaki bildirim de geçerlidir:

```
var d: Dictionary<String, Int> = ["Eskişehir": 26, "İstanbul": 34, "Kocaeli": 41, "Konya": 42]
```

Ya da benzer biçimde:

```
var d: [String: Int]  
  
d = ["Eskişehir": 26, "İstanbul": 34, "Kocaeli": 41, "Konya": 42]  
  
print(d)
```

Dictionary yapısının count property'si sözlükteki eleman sayısını bize vermektedir. Örneğin:

```
var d = ["Eskişehir": 26, "İstanbul": 34, "Kocaeli": 41, "Konya": 42]  
  
print(d.count)    // 4
```

Sözlük türünün en önemli kullanılma nedeni anahtar verildiğinde değeri hızlı bir biçimde (algoritmik yöntemlerle) elde etmektir. Anahtara karşı değer elde edilmesi için Dictionary yapısının subscript elemanı kullanılır. Yapının tek parametrelili subscript elemanında biz köşeli parantezler içerisinde anahtarı verirsek bu subscript bize değeri seçenekselsel olarak (yani V?) türünden verir. Örneğin:

```
var d = ["Eskişehir": 26, "İstanbul": 34, "Kocaeli": 41, "Konya": 42]
```



```

if let value = d["İstanbul"] {
    print(value)
}
else {
    print("anahtar bulunamadı!..")
}

```

Anahtarı vererek değeri elde ettiğimiz tek parametrelili subscript eleman read/write bir elemandır. Sözlüğe eleman ekleme bu subscript elemanın set bölümüyle yapılmaktadır. Örneğin:

```

var d = ["Eskişehir": 26, "İstanbul": 34, "Kocaeli": 41, "Konya": 42]

d["Balıkesir"] = 10
d["Zonguldak"] = 67

print(d)

```

Örneğin:

```

var d: [String : Int] = [:]

d["Ali"] = 123
d["Veli"] = 234
d["Selami"] = 53
d["Ayşe"] = 777
d["Fatma"] = 345

print(d)

```

Aynı anahtar zaten varsa bu işlem o anahtarın değerinin güncellenmesine yol açar. (Örneğin yukarıdaki sözlüğe biz bir tane daha "Ali" anahtarını girersek onun değerini değiştirmiş oluruz. Örneğin:

```

var d = ["Eskişehir": 26, "İstanbul": 34, "Kocaeli": 41, "Konya": 50]

d["Konya"] = 42
print(d)

```

Aynı işlem Dictionary yapısının `updateValue(_:forKey:)` metodu ile de yapılabilir. Metodun birinci parametresi değeri, ikinci parametresi anahtarı alır. Değer varsa onu günceller geri dönüş değeri olarak önceki değeri bize seçeneksel olarak verir. Değer yoksa onu ekler ve nil değerine geri döner. Örneğin:

```

var d = ["Eskişehir": 26, "İstanbul": 34, "Kocaeli": 41, "Konya": 50]

if let oldVal = d.updateValue(42, forKey: "Konya") {
    print("Değer güncellendi, eski değer = \(oldVal)")
}
else {
    print("anahtar/değer eklendi")
}

```

Dictionary yapısının `keys` ve `values` isimli property'leri bize sözlüğün içerisindeki tüm anahtarları ve değerleri bir collection olarak verir. Örneğin:

```

let d = ["Ali": 123, "Veli": 234, "Selami": 532, "Ayşe": 777, "Fatma": 345]

```

```

for key in d.keys {
    print(key, terminator: " ")
}

print()

for value in d.values {
    print(value, terminator: " ")
}

print()

```

Yapının keys ve values property'lerinin bize elemanları sıralı vermediğine dikkat ediniz. Çünkü sözlük organizasyonu anahtar verildiğinde değerin hızlı bir biçimde elde edilmesi amacıyla oluşturulmuştur.

Aslında biz bir sözlüğü de doğrudan for-in döngüsü ile de dolaşabiliriz. Bu durumda her adımda biz (K, V) biçiminde bir demet elde ederiz. Demetler izleyen bölümde ele alınmaktadır. Örneğin:

```

let d = ["Ali": 123, "Veli": 234, "Selami": 532, "Ayşe": 777, "Fatma": 345]

for t in d {
    print("\(t.0), \(t.1)")
}

```

Anahtarı verilen elemanı silmek için removeValue(forKey:) metodu kullanılır. Metot silinen elemanı bize V? biçiminde seçeneksel olarak verir. Örneğin:

```

var d = ["Ali": 123, "Veli": 234, "Selami": 532, "Ayşe": 777, "Fatma": 345]

if let value = d.removeValue(forKey: "Ayşe") {
    print("silinen anahtarın değeri: \(value)")
}
else {
    print("anahtar bulunamadı!")
}

```

Yapının removeAll(keepCapacity:) metodu sözlük içerisindeki tüm elemanları siler. İsteğe bağlı argümanla kapasite korumasının yapıp yapılmayacağı belirlenebilmektedir. Bu parametre default false argümanı almıştır.

Demetler (Tuples)

Demetler özellikle fonksiyonel dillerde çok karşılaştığımız türlerdendir. Pek çok programlama diline de sonraları çeşitli biçimlerde sokulmuştur. Bir demet bir ya da birden fazla değişkenin oluşturduğu topluluktur. Demetler kullanımı kolaylaştırılmış yapılara benzetilebilir. Demetler tipik olarak parantezler içerisinde eleman türleri belirtilerek bildirilir. Örneğin:

```

var t: (Int, String)

```

Burada t ilk elemanı Int, ikinci elemanı String türünden olan bir demettir. Demete değer atama parantezler içerisinde değer listesi belirtilerek yapılır. Başka bir deyişle parantezler içerisinde değerler demet nesnesi anlamına gelmektedir. Örneğin:

```
var t: (Int, String)
```

```
t = (123, "Ali")  
print(t)
```

Tabii tür otomatik olarak da belirlenebilir. Örneğin:

```
let t = (123, "Ali")  
print(t)           // (123, "Ali")  
print(type(of: t)) // (Int, String)
```

Demet elemanlarına nokta operatörü ve elemanın indeks numarasıyla erişilir. İlk eleman sıfırıncı indekstedir, diğerleri bunu izlemektedir. Örneğin:

```
let t = (123, "Ali")  
  
print(t.0) // 123  
print(t.1) // Ali
```

Eğer demet var ile bildirilmişse elemanların değerlerini daha sonra değiştirebiliriz. Örneğin:

```
var t = (123, "Ali")  
  
print(t)  
  
t.0 = 200  
t.1 = "Veli"  
  
print(t)
```

Tabii demetlerin elemanları daha karmaşık türlerden de olabilir. Örneğin:

```
var t = ([1, 2, 3, 4, 5], ["Ali": 10, "Veli": 10, "Selami": 30])  
  
print(type(of: t)) // ([Int], [String: Int])  
  
t.0[2] = 100  
  
print(t) // ([1, 2, 100, 4, 5], ["Selami": 30, "Veli": 10, "Ali": 10])
```

Demetlerin elemanları da demet olabilir. Örneğin:

```
var t = (100, (200, 300))  
  
print(type(of: t)) // (Int, (Int, Int))  
  
print(t) // (100, (200, 300))
```

Demet elemanlarına isimler verilebilir. Bu durumda erişim sırasında bu isimler de kullanılabilir. Örneğin:

```
var pt: (x: Double, y: Double)  
  
pt = (10.2, 4.5)
```

```
print("\(pt.x), \(pt.y)")
print("\(pt.0), \(pt.1)")
```

Tabii demet elemanlarına isim vermiş olsak da yine onlara istersek indeks numarasıyla erişebiliriz. Örneğin:

```
var per: (name: String, no: Int)

per = ("Ali Serçe", 123)
print("\(per.name), \(per.no)")
print("\(per.0), \(per.1)")
```

Tuple elemanlarına isim ilkdeğer verilirken de verilebilir. Örneğin:

```
var pt = (x: 10.2, y: 4.5)

print(pt.x)
print(pt.y)
```

Aslında demetlerdeki eleman isimleri türün bir parçası gibidir. Örneğin:

```
var pt = (x: 10.2, y: 4.5)

print(type(of: pt))    // (x: Double, y: Double)
```

Tabii bir demetin bazı elemanlarına isim verip bazılarını vermeyebiliriz. Örneğin:

```
var x = (name: "Ali", 123, no: 123)

print(type(of: x))    // (name: String, Int, no: Int)
```

Aynı türden ve aynı eleman isimli iki demet birbirine atanabilmektedir. Örneğin:

```
var a: (x: Double, y: Double)
var b: (x: Double, y: Double)

b = (10.2, 3.2)
a = (12.3, 4.5)

print("a = \(a), b = \(b)")
a = b    // geçerli
print("a = \(a), b = \(b)")
```

Örneğin:

```
var a: (x: Double, y: Double)
var b: (z: Double, k: Double)

a = (10.2, 3.2)
b = a    // error! Eleman isimleri aynı değil
```

Örneğin:

```
var a: (x: Double, y: Double)
```

```
a = (z: 10.2, k: 5.3) // error! eleman isimleri aynı değil
```

Örneğin:

```
var a = (name: "Ali Serçe", no: 345)
var b = (name: "Kaan Aslan", no: 123)
```

```
a = b // geçerli
print("a = \(a), b = \(b)")
```

Demet elemanlarına isim verilmemişse isim koşulunun sağlandığı kabul edilir. Örneğin:

```
var a = (x: 10, y: 20)
var b: (Int, Int)
```

```
b = a // geçerli
a = b // geçerli
```

İsimsiz elemanları joker gibi düşünebilirsiniz. Yani biz tür uyumu sağlandıktan sonra isimli bir elemanı isimsiz bir elemana, isimsiz bir elemanı da isimli bir elemana atayabiliriz.

İsimli ve isimsiz demetler birbirlerine atandığında hedef demetin isim durumunda bir değişiklik oluşmamaktadır. Örneğin hedef demet isimsizse biz bu demete isimli bir demet atasak bile bu hedef demet isimsiz olmaya devam edecektir. Benzer biçimde hedef demet isimliyse biz bu demete isimsiz bir demet atasak bile hedef demet isimli olmaya devam edecektir. Örneğin:

```
var a = (x: 10, y: 20)
var b = (30, 40)
```

```
b = a // geçerli
print(b) // (10, 20)
```

Bildirimde tür belirtilmemişse verilen ilkdeğer de bir demetse isimler verilen ilkdeğerden alınmaktadır. Örneğin:

```
var x: (x: Int, y: Int) = (10, 20)
var y = x // dikkat x isimleriyle birlikte atanıyor.
```

```
print(x) // (x: 10, y: 20)
print(y) // (x: 10, y: 20)
```

Demetler let ile bildirilirse elemanların değerleri değiştirilemez. Örneğin:

```
let a = (10, 20.2)
a.0 = 20 // error
a.1 = 12.3 // error
```

Bir demet ters bir işlemle ayrıştırılabilir (decompose edilebilir). Bunun için değişkenlerin isimleri parantezler içerisine yazılarak atama yapılmalıdır. Örneğin:

```
var pt = (10, 20)
var (x, y) = pt // decompose işlemi

print(x) // 10
```

```
print(y)           // 20
```

Bunun eşdeğeri şöyledir:

```
var pt: (Int, Int) = (10, 20)
var x: Int = pt.0, y: Int = pt.1
```

Yukarıdaki ayrıştırma (decompose) işleminin aynı zamanda bir bildirim olduğuna dikkat ediniz. Örneğin:

```
var a = (10, 20)
var (x, y) = a           // decompose işlemi
var x, y: Int           // error! x ve y zaten bildirilmiş
```

Ayrıştırma işlemi bildirim değil atama yoluyla da yapılabilir. Örneğin:

```
var pt = (10, 20)
var x, y: Int

(x, y) = pt
print(x, y)
```

Daha karmaşık ayrıştırmalar da yapılabilir. Örneğin:

```
var a = (10, (30, 40), [1, 2, 3, 4, 5])
let (x, y, z) = a

print(x, y, z)           // 10 (30, 40) [1, 2, 3, 4, 5]
```

Burada x Int, türünden, y (Int, Int) türünden z de [Int] türündendir.

Ayrıştırma işleminde bazı elemanlar istenmiyorsa onun yerine '_' karakteri konulur. Örneğin:

```
var rect = (10, 20, 100, 100)
var (x, _, width, _) = rect

print(x, width)           // 10 100
```

Şüphesiz demetlerden oluşan bir dizi ya da küme söz konusu olabilir. Ya da bir sözlüğün değeri bir demet de olabilir. Örneğin:

```
let a = [(1, 2), (3, 4), (5, 6)]

print(a)
print(a[1])
print(a[1].0)
```

Demetler özellikle bir fonksiyonun birden fazla değerle geri döndürüleceği durumlarda tercih edilmektedir. Örneğin:

```
import Foundation

func getRoots(_ a: Double, _ b: Double, _ c: Double) -> (Double, Double)?
```

```

{
    let delta = b * b - 4 * a * c

    if delta < 0 {
        return nil
    }
    let x1 = (-b + sqrt(delta)) / (2 * a)
    let x2 = (-b - sqrt(delta)) / (2 * a)

    return (x1, x2)
}

var a, b, c: Double

print("a:", terminator: "")
a = Double(readLine()!!)

print("b:", terminator: "")
b = Double(readLine()!!)

print("c:", terminator: "")
c = Double(readLine()!!)

if let roots = getRoots(a, b, c) {
    let (x1, x2) = roots
    print("x1 = \(x1), x2 = \(x2)")
}
else {
    print("kök yok!..")
}

```

Anımsanacağı gibi bir sözlük for-in döngüsüyle dolaşılırken anahtar ve değerler (anahtar, değer) biçiminde demetler olarak elde edilmektedir. Elde edilen demetlerin birinci elemanlarının isimleri "key", ikinci elemanlarının isimleri ise "value" biçimindedir. Örneğin:

```

let dict = ["Ali": 10, "Veli": 20, "Selami": 30, "Ayşe": 40, "Fatma": 50]

for elem in dict {
    print("key = \(elem.key), value = \(elem.value)")
}

```

switch deyiminde de demetler kullanılabilir. Örneğin:

```

let t = (10, 20)

switch t {
    case (1, 2):
        print("bir, iki")
    case (10, _):
        print("on, diğer")
    default:
        print("diğer")
}

```

case ifadelerindeki demetlerde ayrıştırma işlemi de yapılabilir. Örneğin:

```
let t = (10, 20)

switch t {
    case (1, 2):
        print("bir, iki")
    case (let x, let y):
        print("değerler: \(x), \(y)")
}
```

Burada demet ayrıştırılarak case bölümünde elde edilmiştir. Bu kalıp tüm koşulların sağlandığı anlamına gelmektedir.

String Yapısı

String en çok kullanılan yapılardan biridir. Bir String nesnesi String yapısının başlangıç metotlarıyla yaratılabilir. Örneğin boş bir string'i şöyle yaratabiliriz:

```
var str: String = String()
print(str)
```

Aynı işlemi şöyle de yapabiliriz:

```
var str = ""
print(str)
```

Bir String nesnesi String sınıfının Character parametrelili başlangıç metoduyla (init metodu) da yaratılabilir. Örneğin:

```
let ch: Character = "a"
let str: String = String(ch)

print(str)    // a
```

String yapısının init(repeating:count:) biçimindeki başlangıç metodu belli bir karakterden belli bir sayıda olacak biçimde bir String nesnesi oluşturur. Örneğin:

```
let str = String(repeating: "a", count: 10)
print(str)
```

Benzer biçimde bu metodun string alan bir biçimi de vardır. Örneğin:

```
let str = String(repeating: "ankara", count: 5)
print(str)    // ankaraankaraankaraankaraankara
```

String yapısı da Swift 3.0 ile birlikte SequenceType protokolünü destekler hale gelmiştir. Dolayısıyla biz bir String'i for-in döngüsüyle dolaşabiliriz. Bu durumda stringin karakterlerini elde ederiz. Örneğin:

```
let str: String = "Test"

for ch in str {
    print(ch, terminator: "")
}
```



```
print()
```

String yapısının herhangi bir elemanına köşeli parantez operatörüyle erişebiliriz. Ancak köşeli parantez içerisine indeks numarası veremeyiz. Köşeli parantez içerisine indeks belirten özel bir yapı nesnesinin getirilmesi gerekmektedir. İşte String yapısının startIndex property'si ilk karakter için Index değerini, endIndex property'si ise son karakterden bir sonrası için Index değerlerini bize verir. Örneğin:

```
let str = "Test"
let ch = str[str.startIndex]
print(ch)
```

startIndex ve endIndex property'leri

String yapısının startIndex ve endIndex property'leri String yapısının içerisindeki Index isimli bir yapı türündendir. Tabii endIndex bize yazının son karakterinden sonraki indeksi verdiği için biz onu köşeli parantez içerisinde kullanırsak sınır taşmasından dolayı exception oluşur. Örneğin:

```
let str = "Test"
let ch = str[str.endIndex]      // exception oluşur
print(ch)
```

Pekiye herhangi bir elemanın indeksini nasıl elde edebiliriz? Eğer belli bir elemanın indeksini String.Index olarak biliyorsak String yapısının index(after:) index(before) metotlarıyla elde edebiliriz. Örneğin:

```
let str = "Test"
let ch = str[str.index(after: str.startIndex)]
print(ch)      // e
```

String yapısının index metodu yeni bir indeks değeri ile geri dönmektedir. O index değeri öncekinin bir sonraki ya da bir önceki karakterine ilişkindir. Ayrıca biz iki String.Index nesnesini karşılaştırma operatörleriyle karşılaştırabiliriz. Örneğin bir String yapısı bu biçimde bir while döngüsü ile şöyle dolaşılabilir:

```
let str = "ankara"
var index = str.startIndex

while index != str.endIndex {
    print(str[index], terminator: "")
    index = str.index(after: index)
}
print()
```

Aynı biçimde tersten de şöyle yazdırabilirdik:

```
let str = "ankara"
var index = str.endIndex

while index != str.startIndex {
    index = str.index(before: index)
    print(str[index], terminator: "")
}
print()
```

Benzer biçimde String yapısının aynı işlemi yapan ancak yeni indeksi geri dönüş değeri biçiminde değil de argüman olarak geçirilen indeksi değiştirerek veren `formIndex(after:)` ve `formIndex(before:)` metotları da vardır. Bu metotlar inout parametresine sahiptir.

```
let str = "ankara"
var index = str.startIndex

while index != str.endIndex {
    print(str[index], terminator: "")
    str.formIndex(after: &index)
}
print()
```

String yapısının `index(_:offsetBy)` metodu bize geri dönüş değeri olarak mevcut indeksten n ileri ya da geri olan karakterin indeksini verir. Örneğin:

```
let str = "ankara"

let index = str.index(str.startIndex, offsetBy: 4)
print(str[index]) // r
```

Aynı biçimde inout parametrelili `formIndex(_:offsetBy)` metodu da vardır. Bu metot mevcut bir indeksi onun n ilerisi ya da gerisi olacak biçimde günceller.

String yapısının `Range<String.Index>` ve `ClosedRange<String.Index>` parametrelili subscript metotları da vardır. Bu metotlar sayesinde biz köşeli parantezlerin içerisine bir indeks aralığı verebiliriz. Bu işlemten ilgili karakterlerden oluşan yeni bir string elde ederiz. Örneğin:

```
let s = "ankara"

let k = s[s.startIndex...s.endIndex]
print(k) // ankara
```

Örneğin "ankara" yazısının "nkar" kısmını bu yöntemle şöyle elde edebiliriz:

```
let s = "ankara"

let k = s[s.index(s.startIndex, offsetBy: 1)...s.index(s.startIndex, offsetBy: 4)]
print(k) // nkar
```

Bir string'in tüm indekslerini `indices` isimli property elde edebiliriz. `indices` bize dolaşılabilir bir nesne vermektedir. Biz onu for-in döngüsüyle dolaşarak baştan itibaren tüm geçerli index'leri elde edebiliriz. Örneğin:

```
let s = "ankara"

for index in s.indices {
    print(s[index], terminator: "")
}
print()
```

Swift'teki String yapısı C# ve Java'daki String sınıfı gibi "immutable" değildir. Bilindiği gibi o dillerde bir String'in karakterlerini biz değiştiremeyiz. Halbuki Swift'te String nesnesinin karakterleri değiştirilebilmektedir. Swift'in

String yapısı daha çok C++'ın string sınıfına benzemektedir. Swift'teki String yapısını bir çeşit dizi gibi düşünebilirsiniz. Gerçekten de String yapısının içerisinde de bir kapasite tutulmaktadır. String'e ekleme yapıldığında bu kapasite duruma göre artırılmaktadır.

String yapısının append metodu ile yazının sonuna karakter ekleyebiliriz:

```
var s = "eskişehi"
s.append(Character("r"))
print(s)           // eskişehir
```

Benzer biçimde yine String parametrelili append metoduyla da bir yazının sonuna başka bir yazı eklenebilir. Örneğin:

```
var s = "eskişehir"
s.append("ankara")
print(s)           // eskişehirankara
```

Bir String nesnesindeki yazının belli bir karakter öbeğiyle başlayıp başlamadığı, bitip bitmediğini String yapısının hasPrefix ve hasSuffix metotlarıyla belirlenebilir:

```
func hasPrefix(_ prefix: String) -> Bool
func hasSuffix(_ suffix: String) -> Bool
```

```
var s = "eskişehir"

if s.hasPrefix("eski") {
    print("evet")
}
else {
    print("hayır")
}

if s.hasSuffix("şehir") {
    print("evet")
}
else {
    print("hayır")
}
```

String yapısının insert(_at:) metodu bir karakteri belli indekse eklemek için insert(contentsOf:at:) metodu bir yazıyı belli bir indekse eklemek için kullanılmaktadır. Örneğin "eskişehir" 4'üncü indeksine bir karakter eklemek isteyelim:

```
var s = "eskişehir"

s.insert("x", at: s.index(s.startIndex, offsetBy: 4))
print(s)           // eskixşehir
```

Örneğin:

```
var s = "şehir"

s.insert(contentsOf: "eski", at: s.startIndex)
```

```
print(s)           // eskişehir
```

String yapısının removeAll(keepingCapacity:) metodu tüm karakterleri silmek için, remove(at:) metodu ise belli bir indeksteki karakteri silmek için kullanılmaktadır. Örneğin:

```
var s = "eskişehir"

s.remove(at: s.index(s.startIndex, offsetBy: 4))
print(s)           // eskiehir
```

removeAll metodunun parametresi default false değerini almıştır. Yani kapasite koruması yapmaz. Örneğin:

```
var s = "eskişehir"

s.removeAll()
print(s.count)     // 0
```

removeFirst(), removeLast() metotlarının parametresiz ve Int parametrelili overload'ları vardır. Parametresiz biçimler yalnızca baştaki ve sondaki tek karakteri silerler. Int parametreliler baştaki ve sondaki n karakteri silerler. Örneğin:

```
var s = "eskişehir"

s.removeFirst()
print(s)           // skişehir
s.removeLast()
print(s)           // skişehi
s.removeFirst(2)
print(s)           // işehir
s.removeLast(3)
print(s)           // iş
```

Yapının removeSubRange(_:) metodu yazının belli bir aralığını silmekte kullanılabilir. Örneğin:

```
var s = "eskişehir"

s.removeSubrange(s.index(s.startIndex, offsetBy: 2)...s.index(s.startIndex, offsetBy: 6))
print(s)           // esir
```

İki String nesnesini + operatörü ile toplarsak iki yazının bileşiminden oluşan yeni bir String nesnesi elde ederiz. Örneğin:

```
let s = "ankara"
let k = "izmir"

let result = s + k
print(result)
```

Benzer biçimde biz bir String içerisindeki yazıya += operatörüyle de ekleme yapabiliriz. Örneğin:

```
var s = "ankara"
s += "izmir"
```

```
print(s)    // ankaraizmir
```

String yukarıda da vurgulandığı gibi bir yapıdır. Dolayısıyla kategori olarak değer türlerine ilişkindir. Yani bir String türünden nesne karakterlerin kendisini tutmaktadır. Bu durumda biz iki String'i birbirlerine atadığımız zaman birindeki yazı diğerine kopyalanır. Artık birinde yapılan değişiklik diğerini etkilemez. (Halbuki Java ve C#'ta String'ler birer sınıftır. Dolayısıyla String türünden değişkenler nesnenin adresini tutan referanslardır.) Swift'in String yapısının C++'ın string sınıfına bu bakımdan da çok benzediğine dikkat ediniz. Örneğin:

```
var s = "ankara"
var k: String

k = s
s.append(contentsOf:"izmir")
print(s)    // ankaraizmir
print(k)    // ankara
```

String yapısında karakterler default olarak UNICODE biçimde tutulmaktadır. Zaten yapının characters property'si de bize yazının karakterlerini Character türü olarak (yani UNICODE karakter olarak verir). Fakat biz String yapısının utf8 ve utf16 property'leri yoluyla yazının karakterlerini UTF8 ve UTF16 biçiminde elde edebiliriz. utf8 property'si bize yazının karakterlerini UInt8 biçiminde, utf16 ise UInt16 biçiminde verir.

Yapılar (Structures)

Yapılar Swift'te sınıflarla birlikte en önemli tür gruplarından birini oluşturmaktadır. Swift'in Int, Double, String Array<T> gibi temel türleri hep yapı biçiminde organize edilmiştir. Bilindiği gibi Java Programlama Dilinde yapı kavramı yoktur. C++'ta ise yapılarla sınıflar -küçük bir fark dışında- aynı anlamdadır. Swift'e yapılar büyük ölçüde C#'tan esinlenerek aktarılmıştır. Gerçekten de Swift ile C#'ın yapıları işlevsel olarak birbirine çok benzemektedir.

Yapı bildiriminin genel biçimi şöyledir:

```
struct <yapı ismi> {
    <yapı eleman bildirimleri>
}
```

Yapılar kabaca iki tür elemanlara sahip olabilirler: Veri elemanları (properties) ve metotlar (methods). Bir yapı nesnesinin yaratılması şöyle yapılmaktadır:

```
<yapı ismi>([argüman listesi])
```

Yapı nesnesi yaratıldığında nesneye ilkdeğer vermek için ilkdeğerleme metodu (initializer) çağrılmaktadır. Swift'te ilkdeğerleme metodunun ismi init olmak zorundadır. init metot bildirimlerinin genel biçimi şöyledir:

```
[erişim belirleyicisi] init([parametre değişken bildirimleri])
{
    //...
}
```

init metodunun bildiriminde func anahtar sözcüğünün kullanılmadığına dikkat ediniz. Diğer dillerde olduğu gibi Swift'te de ilkdeğerleme metotlarının geri dönüş değerleri diye bir kavramları yoktur. İlkdeğerleme metodunun

amacı belli birtakım ilk işlemleri yapmak ve yapının özelliklerine (veri elemanlarına) bazı ilkdeğerleri vermektir. Örneğin:

```
struct Rect {
    var x: Double
    var y: Double
    var width: Double
    var height: Double

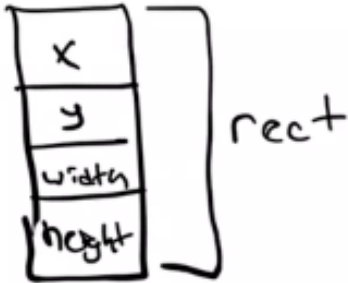
    init()
    {
        x = 0
        y = 0
        width = 0
        height = 0
    }

    func disp()
    {
        print("x = \(x), y = \(y), width = \(width), height = \(height)")
    }
}

var rect = Rect()

rect.disp()
```

Yapılar bileşik türlerdir. Yani parçalardan oluşmaktadır. Yapıların veri elemanları (property'leri) bu parçaları belirtir. Örneğin yukarıdaki gibi bir Rect yapısı türünden bir nesne yaratıldığında bu nesne kendi içerisinde x, y, width ve height biçiminde dört double türden parçaya sahip olur. Yapının veri elemanları yapının metotları tarafından ortak olarak kullanılırlar. Yani biz yapının metot içerisinde yapının veri elemanlarını doğrudan kullanabiliriz. Daha önceden de belirtildiği gibi yapının metotları yapı nesnesi içerisinde yer kaplamazlar. Bunlar yalnızca mantıksal olarak yapı ile ilişkilendirilmiştir.



Yukarıdaki örneğimizde rect global bir nesnedir. Dolayısıyla dört parçası ile birlikte Data/BSS alanında yaratılır. Aslında bir yapı nesnesinin yaratılması şöyle yapılmaktadır:

```
var rect = Rect()
```

yapı ismini biz parantezle kullandığımızda o noktada geçici bir yapı nesnesi oluşturulur. Sonra geçici yapı nesnesinin elemanları asıl yapı nesnesine atanmaktadır. Bu geçici nesne de bu işlem sonucunda yok edilmektedir. Tabii derleyiciler bu konuda işlevler aynı kalmak üzere optimizasyonlar yapabilirler. Bildirimden sonra yine bir yapı değişkenine bu biçimde değer atayabiliriz:

```
rect = Rect()
```

Burada yine Rect() işlemi ile geçici bir yapı nesnesi yaratılmıştır. Bu nesnenin karşılıklı elemanları rect nesnesine atanmıştır. Sonra bu geçici nesne ifade bittiğinde derleyici tarafından yok edilecektir.

Bir yapı türünden nesnenin ismi bu nesnenin tamamını temsil eder. Yapının belli bir veri elemanına erişip onu bağımsız bir nesne gibi de kullanabiliriz. Bunun için nokta operatörü kullanılmaktadır. Örneğin:

```
var rect: Rect = Rect()
print("x = \(rect.x), y = \(rect.y), width = \(rect.width), height = \(rect.height)")
```

Swift'te sınıfların ve yapıların dışındaki alt programlara "fonksiyon (function)", sınıfların ve yapıların içerisindeki alt programlara ise "metot (method)" denilmektedir. Bir metod ilgili sınıf ya da yapı türünden bir değişkenle ve nokta operatörü kullanılarak çağrılır. Örneğin:

```
rect.disp()
```

Swift'te yapı nesneleri yaratıldığında (tıpkı C#'ta olduğu gibi) onların bütün veri elemanlarına (properties) değer atanmış olmak zorundadır. Örneğin:

```
struct Rect {
    var x: Double
    var y: Double
    var width: Double
    var height: Double

    init() // error! width ve height property'lerine değer atanmamış
    {
        x = 0
        y = 0
    }
}
```

Yapının veri elemanlarına değer atamanın iki yolu vardır: Bildirim sırasında ilkdeğer verme biçiminde değer atamak ya da init metodu içerisinde değer atamak. Örneğin:

```
struct Rect {
    var x: Double
    var y: Double
    var width: Double = 10
    var height: Double = 10

    init() // geçerli! tüm veri elemanlarına değer atanmış
    {
        x = 0
        y = 0
    }
}
```

Eğer property'lere her iki biçimde de değer atanırsa init metodunun içerisinde atanmış olan değerler onların içerisinde kalacaktır. Çünkü Swift derleyicileri yapı bildiriminde verilen ilkdeğerleri atama deyimlerine dönüştürerek yapının tüm init metodlarının ana bloğunun başına yerleştirmektedir. Örneğin:

```

struct Rect {
    var x: Double = 0
    var y: Double = 0
    var width: Double = 0
    var height: Double = 0

    init()
    {
        x = 10
        y = 10
    }

    func disp()
    {
        print("x = \(x), y = \(y), width = \(width), height = \(height)")
        // x = 10.0, y = 10.0, width = 0.0, height = 0.0
    }
}

var rect: Rect = Rect()

rect.disp()

```

Swift'te de diğer dillerde olduğu gibi eğer programcı yapı için hiçbir init metodu yazmamışsa ve yapının tüm elemanlarına bildirim sırasında ilkdeğer vermişse derleyici parametresiz init metodunu (yani default başlangıç metodunu) içi boş olarak bizim için kendisi yazmaktadır. Örneğin:

```

struct Rect {
    var x: Double = 0
    var y: Double = 0
    var width: Double = 10
    var height: Double = 10

    func disp()
    {
        print("x = \(x), y = \(y), width = \(width), height = \(height)")
    }
}

var rect: Rect = Rect()    // geçerli
rect.disp()

```

Fakat örneğin:

```

struct Rect {
    var x: Double = 0
    var y: Double = 0
    var width: Double
    var height: Double

    func disp()
    {
        print("x = \(x), y = \(y), width = \(width), height = \(height)")
    }
}

```



```

}

var rect: Rect = Rect()    // error!
rect.disp()

```

Parametresiz init metodunun yanı sıra yine Swift'te programcı yapı için hiçbir init metodu yazmamışsa derleyici yapı için eleman ilkdeğerlemesi yapan (memberwise initialization) bir init metodunu da kendisi yazmaktadır. Örneğin:

```

struct Rect {
    var x: Double = 0
    var y: Double = 0
    var width: Double
    var height: Double

    func disp()
    {
        print("x = \(x), y = \(y), width = \(width), height = \(height)")
    }
}

var rect: Rect = Rect(x: 10, y: 10, width: 100, height: 100)
rect.disp()

```

Burada derleyici aşağıdaki gibi bir init metodunu kendisi yazmaktadır:

```

init(x: Double, y: Double, width: Double, height: Double)
{
    self.x = x
    self.y = y
    self.width = width
    self.height = height
}

```

Eleman ilkdeğerlemesi yapan init metotlarında argümanların veri eleman bildirim sırasına göre girilmesi zorunludur. Örneğin biz nesneyi aşağıdaki gibi yaratamazdık:

```

var rect: Rect = Rect(y: 10, x: 10, width: 100, height: 200)    // error!

```

Biz bir yapının bütün veri elemanlarına yapı bildirimi sırasında ilkdeğer vermiş olalım. Fakat yapı için de hiçbir init metodu da yazmamış olalım. İşte derleyici bu durumda hem eleman ilkdeğerlemesi yapan init metodunu hem de parametresiz init metodunu bizim için yazacaktır. Örneğin:

```

struct Rect {
    var x: Double = 0
    var y: Double = 0
    var width: Double = 100
    var height: Double = 100

    func disp()
    {
        print("x = \(x), y = \(y), width = \(width), height = \(height)")
    }
}

```

```
var rect1: Rect = Rect()
rect1.disp()

var rect2: Rect = Rect(x: 10, y: 10, width: 50, height: 50)
rect2.disp()
```

Yukarıdaki anlatımları şöyle özetleyebiliriz:

- 1) Programcı yapı için herhangi bir init metodu yazmışsa derleyici programcı için hiçbir init metodu yazmaz.
- 2) Programcı yapı için hiçbir init metodu yazmamışsa derleyici eleman ilkdeğerlemesi yapan init metodunu kendisi yazar. Ayrıca derleyici parametresiz init metodunu da içi boş olarak programcı eğer yapının tüm elemanlarına bildirim sırasında ilkdeğer verilmişse kendisi yazmaktadır.

init metotları overload edilebilir. Yani yapının parametrik yapısı ya da etiketleri farklı olan birden fazla init metodu olabilir. Biz yine bunlarda etiket isimleri için '_' belirlemesi yapmamışsak her zaman etiket isimlerini argüman listesinde belirtmek zorundayız. Örneğin:

```
struct Point {
    var x: Double
    var y: Double

    init()
    {
        x = 0
        y = 0
    }

    init(x: Double, y: Double)
    {
        self.x = x
        self.y = y
    }

    func disp()
    {
        print("x = \(x), y = \(y)")
    }
}

var pt: Point = Point(x: 10, y: 20)
pt.disp()
```

Swift'te yapının bir init metodu başka bir init metodunu çağırabilir. Bu duruma Swift dokümanlarında "initializer delegation" denilmektedir. Ancak çağırma işleminin self anahtar sözcüğü ile yapılması zorunludur. Örneğin:

```
struct Point {
    var x: Double
    var y: Double

    init(x: Double, y: Double)
    {
        self.x = x
    }
}
```

```

        self.y = y
    }

    init()
    {
        self.init(x: 0, y: 0)
    }

    func disp()
    {
        print("x = \(x), y = \(y)")
    }
}

var pt: Point = Point()
pt.disp()

```

Yapı nesneleri let ile bildirilirse onun bütün veri elemanları let gibi olur. Yani init metodunun dışında onlara değer atayamayız. Örneğin:

```

let date: Date = Date(day: 10, month: 12, year: 2009)

date.day = 11 // error!

```

let ile bildirilmiş yapı nesneleriyle yapının mutating metotları çağrılmaz. Mutating metotlar ileride ele alınmaktadır.

Yapıların belli elemanları da let ile bildirilebilir. Bu durumda yapı nesnesi var ile bildirilmiş olsa bile biz let ile bildirilen elemanların değerlerini değiştiremeyiz. let ile bildirilmiş veri elemanlarına yapı bildiriminde ya da init metotlarında ilkdeğerleri verilebilir. Bunun dışında bu elemanlara değer atayamayız. Örneğin:

```

struct Point {
    let x: Double
    var y: Double

    init(x: Double, y: Double)
    {
        self.x = x // geçerli, init'te atama yapılabilir
        self.y = y
    }

    func disp()
    {
        print("\(x), \(y)")
    }
}

var pt = Point(x: 10, y: 12)
pt.x = 20 // error!

```

Yapıların metotları default durumda yapıların veri elemanlarını değiştiremez. (Yani bunlar "C++'taki const üye fonksiyonlar" gibidir.) Örneğin:

```

struct Point {
    var x: Double

```

```

var y: Double

init(x: Double, y: Double)
{
    self.x = x        // geçerli, init'te atama yapılabilir
    self.y = y
}

func foo()
{
    print("(x), (y)")    // geçerli
    x = 100              // error!
    y = 200              // error!
}
}

```

Yapının belli bir metodunun veri elemanlarının değerlerini değiştirmesini istiyorsak onu mutating yapmamız gerekir. Örneğin:

```

struct Point {
    var x: Double
    var y: Double

    init(x: Double, y: Double)
    {
        self.x = x        // geçerli, init'te atama yapılabilir
        self.y = y
    }

    mutating func foo()
    {
        print("(x), (y)")    // geçerli
        x = 100              // geçerli
        y = 200              // geçerli
    }
}

```

mutating anahtar sözcüğünün metodun başına getirildiğine dikkat ediniz. Global fonksiyonlar mutating yapılamazlar. Tabii mutating metotlar yine yapının let veri elemanlarını değiştiremezler.

Yapılar kategori olarak değer türlerine ilişkindir. Yani bir yapı türünden değişken bileşik bir nesnedir ve doğrudan yapı elemanlarının değerlerini tutar. Örneğin:

```

struct Date {
    var day: Int
    var month: Int
    var year: Int

    init(day: Int, month: Int, year: Int)
    {
        self.day = day
        self.month = month
        self.year = year
    }

    func disp()

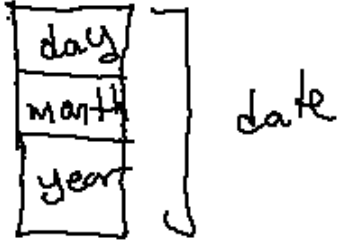
```

```

{
    print("\(day)/\(month)/\(year)")
}
}

var date: Date = Date(day: 10, month: 12, year: 2009)
date.disp()

```



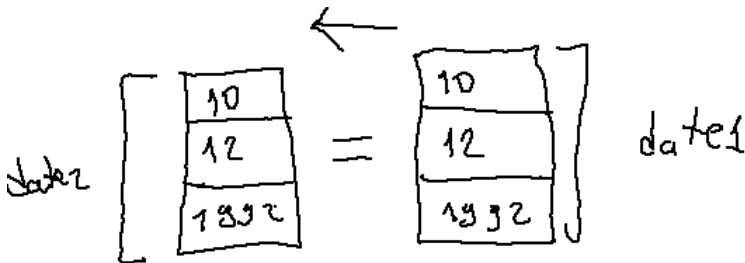
Şüphesiz derleyici yapı elemanlarını ardışıl bir biçimde tutar. Ancak dilde resmi olarak böyle bir garanti verilmemiştir.

Aynı türden iki yapı değişkeni birbirlerine atanabilir. Bu durumda onların bütün veri elemanları birbirlerine atanacaktır. Örneğin:

```

var date1: Date = Date(day: 12, month: 10, year: 2001)
var date2: Date
date2 = date1

```



Yapı değişkenlerinin fonksiyonlara ve metotlara parametre yoluyla aktarılması da kopyalama ile (call by value) yapılmaktadır. Ancak derleyici diziler konusunda da ele alındığı gibi arka planda birtakım optimizasyonlar yapabilmektedir. Yani yapı nesnesini arka planda adres yoluyla da fonksiyonu ya da metoda geçirebilmektedir. Örneğin:

```

struct Date {
    var day: Int
    var month: Int
    var year: Int

    func disp()
    {
        print("\(day)/\(month)/\(year)")
    }
}

func foo(date: Date)
{

```

```

    date.disp()
}

var endOfTheYear: Date = Date(day: 31, month: 12, year: 2018)
foo(date: endOfTheYear)

```

Anımsanacağı gibi Swift'te inout olmayan parametreler üzerinde fonksiyonlar ya da metotlar değişiklik yapamıyorlardı. Yani parametre değişkenleri Swift'te default olarak let durumundadır. İşte bir fonksiyonun ya da metodun parametresi bir yapı türündense bu fonksiyon ya da metot yapının hiçbir elemanını değiştirmez. Bu durumda derleyici aslında yapıları her zaman arka planda optimizasyon amaçlı fonksiyon ya da yapılara adres yoluyla aktarmaktadır. Örneğin:

```

struct Date {
    var day: Int
    var month: Int
    var year: Int

    func disp()
    {
        print("\(day)/\(month)/\(year)")
    }
}

func foo(date: inout Date)
{
    date.disp()

    date.day = 1
    date.month = 1
    date.year = 1900
}

var endOfTheYear: Date = Date(day: 31, month: 12, year: 2018)
foo(date: &endOfTheYear)
endOfTheYear.disp()

```

Tabii yapı nesnesinin kendisi let ise biz bunu inout parametresiyle fonksiyona ya da metoda aktaramayız. Ancak yapı nesnesinin kendisi var ancak bazı elemanları let ise biz bu nesneyi inout parametresi ile fonksiyonlara ya da metotlara aktarabiliriz. Ancak bu durumda fonksiyon ya da metot yapının let elemanlarını değiştiremeyecektir.

Yapılarda mutating Metotlar

Swift'te default durumda kategori olarak değer türlerine (value types) ilişkin veri yapılarının metotları (yani struct ve enum metotları) ilgili veri yapısının veri elemanlarını (property'lerini) değiştiremez. Yapıların ve enum türlerinin metotlarının kendi veri elemanlarını değiştirebilmesi için metot bildiriminin başında mutating anahtar sözcüğünün bulundurulması gerekir. Örneğin:

```

struct Sample {
    var a: Int
    var b: Int

    mutating func foo()
    {

```

```

    a = 10          // error!
    b = 20          // error
}

mutating func bar()
{
    a = 10          // geçerli
    b = 20          // geçerli
}

func tar()
{
    print(a, b)     // geçerli
}
}

```

Tabii bir yapı ya da enum türünden bir let değişken ile o yapının ya da enum türünün mutating bir metodunu çağıramayız. Örneğin:

```

struct Sample {
    var a: Int
    var b: Int

    init()
    {
        a = 0
        b = 0
    }

    func foo()
    {
        print(a)
        print(b)
    }

    mutating func bar()
    {
        a = 10      // geçerli
        b = 20      // geçerli
    }
}

let s = Sample()

s.foo()           // geçerli
s.bar()           // error!

```

Yapıların Property Elemanları (Computed Properties)

Swift'te "property" sözcüğü hem veri elemanları için hem de onlara erişen get/set metotları için ortak kullanılan bir terim belirtmektedir. Fakat Swift'te veri elemanları için depolanmış property (stored property), erişimciler için ise hesaplanmış property ("computed property") terimi de kullanılmaktadır.

Hesaplanmış property'ler C#'taki normal property'ler gibidir. Zaten bunların sentaksı büyük ölçüde C#'tan alınmıştır. Hesaplanmış property'lerin genel biçimi şöyledir:

```

var <property ismi>: <tür> {
    get {
        //...
    }
    set [(<parametre ismi>)] {
        //...
    }
}

```

Hesaplanmış bir property değer atamak amaçlı olarak bir ifadede kullanılmışsa property'nin set bölümü çalıştırılır, atanacak değer de set bölümüne argüman olarak aktarılır. Eğer hesaplanmış property değer alma amaçlı olarak bir ifadede kullanılmışsa property'nin get bölümü çalıştırılır. Get bölümünden return edilen değer property'nin değeri olarak elde edilir.

Hesaplanmış property'ler let anahtar sözcüğü ile bildirilemezler, yalnızca var anahtar sözcüğü ile bildirilebilirler. Fakat bir hesaplanmış property yalnızca get bölümüne (read-only) ya da hem get hem de set bölümüne (read/write) sahip olabilir. Fakat Swift'te C#'taki gibi yalnızca set bölümüne sahip (write-only) property yazılamamaktadır.

Swift'te hesaplanmış property'ler aslında var olmayan veri elemanlarının varmış gibi sunulması amacıyla kullanılmaktadır. (Zaten hesaplanmış ismi de buradan geliyor). Örneğin:

```

import Foundation

struct Point {
    var x: Double = 0
    var y: Double = 0

    func disp()
    {
        print("x = \(x), y = \(y)")
    }
}

struct Size {
    var width: Double
    var height: Double

    func disp()
    {
        print("width = \(width), height = \(height)")
    }
}

struct Rectangle {
    var pt: Point
    var sz: Size

    var center: Point {
        get {
            return Point(x: pt.x + sz.width / 2, y: pt.y + sz.height / 2)
        }
    }
}

```



```

        set (newCenter) {
            pt.x = newCenter.x - sz.width / 2
            pt.y = newCenter.y - sz.height / 2
        }
    }

    func disp()
    {
        print("pt = \(pt.x), \(pt.y)), sz = \(sz.width), \(sz.height)")
    }
}

var rect: Rectangle = Rectangle(pt: Point(x: 10, y: 10), sz: Size(width: 20, height: 20))
rect.disp()

rect.center.disp()           // get bölümü çalıştırılır
rect.center = Point(x: 10, y: 10) // set bölümü çalıştırılır
rect.disp()

```

Bu örnekte center isimli bir veri elemanı yoktur. Yani dikdörtgenin merkez noktası yapının içerisinde yer kaplayan bir değişkenle tutulmamıştır. Programcı center property'sini kullandığında dikdörtgenin merkez koordinatları sol-üst köşe koordinatından hareketle hesap yapılarak elde edilmektedir.

Hesaplanmış property'nin parametre ismi belirtilmezse default olarak newValue anahtar sözcüğünün parametre ismi olarak belirlendiği kabul edilir. Örneğin:

```

var center: Point {
    get {
        return Point(x: (pt.x + sz.width) / 2, y: (pt.y + sz.height) / 2)
    }
    set {
        pt.x = newValue.x - sz.width / 2
        pt.y = newValue.y - sz.height / 2
    }
}

```

Örneğin:

```

import Foundation

struct Square {
    var edgeLen: Double

    var area: Double {
        get {
            return edgeLen * edgeLen
        }
        set {
            edgeLen = sqrt(newValue)
        }
    }
}

init()
{
    edgeLen = 0
}

```

```

init(edgeLen: Double)
{
    self.edgeLen = edgeLen
}

func disp()
{
    print("edge length = \(edgeLen), area = \(area)")
}
}

var s = Square(edgeLen: 10)
s.disp()
print(s.area)
s.area = 20
s.disp()

```

Read-only hesaplanmış property'lerde get bölümü de hiç belirtilmeyebilir. Örneğin:

```

var center: Point {
    return Point(x: (pt.x + sz.width) / 2, y: (pt.y + sz.height) / 2)
}

```

Bu property bildirimi aşağıdaki ile eşdeğerdir:

```

var center: Point {
    get {
        return Point(x: pt.x + sz.width / 2, y: pt.y + sz.height / 2)
    }
}

```

Örneğin:

```

import Foundation

struct Complex {
    var real: Double
    var imag: Double

    init()
    {
        real = 0
        imag = 0
    }

    init(real: Double, imag: Double)
    {
        self.real = real
        self.imag = imag
    }

    var magnitude: Double {
        return sqrt(real * real + imag * imag)
    }

    func disp()

```

```

    {
        print("\(real)+\(imag)i")
    }
}

let z = Complex(real: 3, imag: 2)
z.disp()
print(z.magnitude)

```

Peki hesaplanmış property'lere neden gereksinim duyulmaktadır? Bunun iki nedeni vardır:

1) private bölüme yerleştirilmiş olan veri elemanlarına erişmek için. Bu konu ileride ele alınacaktır.

2) Başka birtakım veri elemanları üzerinde işlem yapılarak değeri hesaplanacak özellikler için. Örneğin Rectangle yapısında dikdörtgenin merkez koordinatlarını tutmaya gerek yoktur. Zaten bu koordinatlar istenildiğinde diğer veri elemanlarından hareketle hesaplanabilmektedir. Benzer biçimde Square sınıfında da karenin ayrıca alanının tutulmasına gerek yoktur.

Şimdiye kadar gördüğümüz bazı yapıların bazı elemanları aslında hesaplanmış property'lerdir. Örneğin Array yapısının ve String yapısının count elemanları aslında birer hesaplanmış property'dir.

Property Gözlemcileri (Property Observers)

Depolanmış bir property'ye değer atamadan az önce ve atandıktan hemen sonra bir kodun çalıştırılmasını sağlayabiliriz. Bu kodlara "property gözlemcileri (property observers)" denilmektedir. Property gözlemci bildiriminin genel biçimi şöyledir:

```

<property bildirimi> {
    willSet [(<parametre ismi>)] {
        //...
    }
    didSet [(<parametre ismi>)] {
        //...
    }
}

```

Property'ye değer atanmadan hemen önce gözlemcinin willSet bölümü, property'ye değer atandıktan hemen sonra da didSet bölümü çalıştırılır. Property'ye atanacak değer willSet bölümüne parametre olarak geçirilmektedir. Eğer willSet bölümünde parametre ismi belirtilmezse newValue ismi parametre ismi olarak kullanılır. Benzer biçimde didSet bölümündeki parametre de property'nin değiştirilmeden önceki değerini belirtir. didSet bölümünde parametre ismi belirtilmezse oldValue ismi parametre ismi olarak kullanılır. Property değişkeninin ismi hem willSet bölümünde hem de didSet bölümünde kullanılabilir. Örneğin:

```

struct Point {
    var x: Double = 0 {
        willSet (newX) {
            print("willSet(x): \(newX), \(x)")
        }
        didSet (oldX) {
            print("didSet(x): \(oldX), \(x)")
        }
    }
}

```

```

var y: Double = 0 {
    willSet (newY) {
        print("willSet(y): \(newY), \(y)")
    }
    didSet (oldY) {
        print("didset(y): \(oldY), \(y)")
    }
}

init(x: Double, y: Double)
{
    self.x = x
    self.y = y
}

func disp()
{
    print("x = \(x), y = \(y)")
}
}

```

```

var pt: Point = Point(x: 100, y: 100)
pt.x = 200
pt.y = 200

```

Property gözlemcilerinin yalnızca willSet bölümü ya da yalnızca didSet bölümü bulundurulabilir. Eğer her iki bölüm de bulundurulacaksa bunlar herhangi bir sırada bildirilebilirler (yani önce willSet sonra didSet ya da önce didSet sonra willSet).

Property gözlemcileri property değişkenine ilkdeğer verilirken çalıştırılmazlar. init metotları içerisinde onlara değer atanırken de çalıştırılmazlar. Ancak bunun dışındaki değer atamalarda (örneğin bir metot içerisinde değer atamalarda ya da dışarıdan değer atamalarda) çalıştırılırlar.

Yapıların static Elemanları

Swift'te de diğer nesne yönelimli dillerde olduğu gibi yapılar ve sınıflar static elemanlara (yani property'lere ve metotlara) sahip olabilirler. Swift'te sınıfın statik veri elemanlarına "tür property'leri (type properties)" denilmektedir. Tür property'leri diğer pek çok dilde olduğu gibi Swift'te de static anahtar sözcüğü ile bildirilmektedir. Tür property'lerine (yani statik veri elemanlarına) bildirim sırasında ilkdeğer verilmesi zorunludur. Tür proeprty'lerinin toplamda tek bir kopyası vardır.

Swift'te tıpkı C#'ta olduğu gibi yapıların static elemanlarına sınıf ismi ile erişilir. Ancak C++, Java ve C#'tan farklı olarak Swift'te yapıların static olmayan metotları static elemanları doğrudan isimleriyle (yani nitelsiz olarak) kullanamazlar. Yani static olmayan metotlar (instance metotlar) yapıların static elemanlarına yapı ismi ve nokta operatörü ile erişebilirler. Örneğin:

```

struct Sample {
    var a: Int
    var b: Int
    static var count: Int = 0

    init()
    {
        a = 0
    }
}

```

```

        b = 0
        ++Sample.count
        // ++count ---> error!
    }
}

var x = Sample()
var y = Sample()
var z = Sample()

print(Sample.count)

```

Burada init static olmayan bir metot durumundadır. Bu nedenle static count property'sine tür ismiyle erişilmiştir. Fakat yapının static elemanları static metotlar içerisinde hiç tür ismi belirtilmeden yani niteliksiz olarak kullanılabilir. Örneğin:

```

struct Sample {
    var a: Int
    var b: Int
    static var count: Int = 0

    init()
    {
        a = 0
        b = 0
        ++Sample.count
    }

    static var negCount: Int {
        get {
            return -count
        }
        set {
            count = -newValue
        }
    }

    static func dispCount()
    {
        print(count)    // print(Sample.count) da olabilirdi
    }
}

var x = Sample()
var y = Sample()
var z = Sample()

print(Sample.negCount)
Sample.negCount = -10
Sample.dispCount()

```

Örneğin:

```

import Foundation

struct Date {
    var day: Int

```

```

var month: Int
var year: Int
static var monTab = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
static var dayText = ["Pazar", "Pazartesi", "Salı", "Çarşamba", "Perşembe", "Cuma",
"Cumartesi"]

init()
{
    day = 1
    month = 1
    year = 1900
}

init(day: Int, month: Int, year: Int)
{
    self.day = day
    self.month = month
    self.year = year
}

static func isLeapYear(year: Int) -> Bool
{
    return year % 400 == 0 || year % 4 == 0 && year % 100 != 0
}

func getTotalDays() -> Int
{
    var total = 0

    for i in 1900..

```

Yapıların static metotlarında self anahtar sözcüğü kullanılabilir. Ancak burada self anahtar sözcüğü türü belirtiyor durumdadır metodun çağrıldığı nesneyi değil. Örneğin:

```

struct Sample {
    static var x: Int = 10
    static func disp()
    {
        print(self.x)      // print(Sample.x)
    }
    //...
}

```

Diğer nesne yönelimli dillerde olduğu gibi yapının static metotları yapının yalnızca static elemanlarını, yapının static olmayan metotları ise yapının hem static elemanlarını hem de static olmayan elemanlarını kullanabilir. Ancak static olmayan metotlar yapının static elemanlarına doğrudan değil yapı ismiyle niteliklendirerek erişebilmektedir.

Swift'te aynı yapı ya da sınıfta aynı isimli aynı parametrik türlere ve etiket isimlerine ve aynı geri dönüş değeri türüne sahip static olan ve static olmayan iki metot birarada bulunabilir. (Anımsanacağı gibi C++, Java ve C#'ta bu mümkün değildir.) Bunun Swift'te mümkün olması static olmayan metotlardan static elemanların yapı ya da sınıf ismiyle kombine edilerek kullanılabilmesindendir. Örneğin:

```

struct Sample {
    var a: Int = 0

    func foo()
    {
        //...
    }

    static func foo()      // geçerli
    {
        //...
    }

    func bar()
    {
        foo()              // static olmayan (instance) foo
        Sample.foo()       // static olan foo
    }

    static func tar()
    {
        foo()              // static olan foo
    }
}

var s = Sample()

s.foo()                  // static olmayan (instance) foo
Sample.foo()             // static foo

```

Yapıların Seçeneksel Veri Elemanları

Yapıların seçeneksel (optional) elemanlarına ilkdeğer verilmeyebilir. Bu durumda bu elemanların içerisine derleyici tarafından nil değeri atanmaktadır. Örneğin:

```

struct Sample {
    var a: Int
    var b: Int?

    init()
    {
        a = 10
    }
}

var s = Sample()

print(s.a)                // 10
print(s.b == nil ? "nil" : s.b!) // nil

```

Burada biz init metodu içerisinde seçeneysel b property'sine değer atamadık. Fakat bunun içerisinde default değer olarak nil bulunacaktır.

Yapıların Subscript Elemanları

Bir yapı (ya da sınıf) türünden değişkenin köşeli parantez operatörüyle kullanılabilmesi için o yapının (ya da sınıfın) subscript isimli bir elemanının olması gerekir. Swift'teki subscript elemanlar C#'taki indeksleyicilerin işlevsel olarak eşdeğeridir. Zaten bu özellik Swift'e -hesaplanmış property'lerle birlikte- C#'tan aktarılmıştır. Bilindiği gibi sınıf ya da yapı nesnelerinin köşeli parantezlerle kullanımı C++'ta doğrudan operatör fonksiyonlarıyla yapılmaktadır.

Subscript bildiriminin genel biçimi şöyledir:

```

subscript ([parametre bildirimi]) -> <geri dönüş değerinin türü>
{
    get {
        //...
    }
    set [(<parametre ismi>)] {
        //...
    }
}

```

s bir yapı türünden nesne olsun. Bu yapı nesnesini köşeli parantezlerle iki amaçla kullanabiliriz: Değer almak ve değer yerleştirmek. Örneğin:

```

s[i] = val           // burada subscript değer yerleştirme amacıyla kullanılmıştır
val = s[i]           // burada subscript değer alma amacıyla kullanılmıştır

```

Yapı nesnesi köşeli parantez operatörleriyle değer almak amaçlı kullanılıyorsa subscript'in get bölümü, değer yerleştirmek amaçlı kullanılıyorsa set bölümü çalıştırılmaktadır.

Şüphesiz bir yapının subscript elemanının yazılabilmesi için yapının collection benzeri bir niteliğe sahip olması gerekir. Örneğin:

```

struct ArrayWrapper {

```



```

var array: [Int]

init(array: [Int])
{
    self.array = array
}

subscript (index: Int) -> Int {
    get {
        return array[index]
    }
    set {
        array[index] = newValue
    }
}

var count: Int {
    return array.count
}
}

var aw = ArrayWrapper(array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

for i in 0..

```

Subscript elemanların sentaks ve semantik özellikleri de hesaplanmış property'lerle aynı biçimdedir. Subscript'lerin de set bölümünde parametre ismi verilmezse default parametre ismi newValue biçimindedir.

Subscript'ler de read-only ya da read/write olabilir. Ancak write-only olamazlar. Read-only subscript'lerde get bölümü hiç belirtilmeyebilir. Yani örneğin:

```

subscript (a: Int) -> Int {
    get {
        //...
    }
}

```

bildirimi ile:

```

subscript (a: Int) -> Int {
    //...
}

```

bildirimi eşdeğerdir.

Örneğin:

```
struct CumulativeArray {
    var array: [Int]

    subscript (index: Int) -> Int {
        var total = 0
        for var i = 0; i <= index; ++i {
            total += array[i]
        }
        return total
    }
}

var ca = CumulativeArray(array: [1, 2, 3, 4, 5, 6, 7, 8, 9])

print(ca[3])
print(ca[7])
```

Subscript'ler birden fazla parametreye sahip olabilir. Böyle subscriptler köşeli parantez içerisinde birden fazla değer girilerek kullanılırlar. Örneğin bir matrisi temsil eden bir sınıf şöyle yazılabilir:

```
struct IntMatrix {
    var array: [Int]
    let rowSize: Int
    let colSize: Int

    init(rowSize: Int, colSize: Int)
    {
        self.rowSize = rowSize
        self.colSize = colSize
        array = Array(repeating: 0, count: rowSize * colSize )
    }

    subscript (row: Int, col: Int) -> Int {
        get {
            return array[row * colSize + col]
        }
        set {
            array[row * colSize + col] = newValue
        }
    }
}

var im = IntMatrix(rowSize: 3, colSize: 3)

for i in 0..<3 {
    for k in 0..<3 {
        im[i, k] = i + k
    }
}

for i in 0..<3 {
    for k in 0..<3 {
        print(im[i, k], terminator: " ")
    }
}
```

```
    print()
}
print()
```

Subscript metotları da overload edilebilir. Yani yapının farklı parametre sayılarına, farklı türlere ya da farklı geri dönüş değerlerine sahip birden fazla subscript elemanı bulunabilir. Swift'te de static subscript elemanlar bildirilememektedir.

Swift'te Enum Türleri ve Sabitleri

Enum türleri pek çok dilde benzer biçimde bulunmaktadır. Enum türlerinden amaç kısıtlı sayıda seçeneğe sahip olan olguları hem sayısal düzeyde hem isimlerle nitelemektir. Örneğin haftanın günleri, yönler, renkler gibi olgular tipik olarak enum türleriyle temsil edilebilmektedir.

Swift'in enum türleri semantik bakımdan C, C++ ve C#'tan ziyade Java'ya benzemektedir. Bir enum bildirimi sıfır ya da daha fazla enum sabiti içerebilir. Swift'te enum türleri protokoleri destekleyebilir. Ancak yapılarda olduğu gibi enum türleri de türetmeye kapalıdır. Yani bir enum'dan türetme yapılamaz bir enum da başka bir türden türetilemez. Enum türleri kategori olarak değer türlerine (value types) ilişkindir. Yani enum türünden bir değişken bir adres değil değerini tutar. Enum bildiriminin genel biçimi şöyledir:

```
enum <isim> [ : <protokol istesi> ] {
    [enum sabit listesi]
}
```

enum sabit bildiriminin ise genel biçimi şöyledir:

```
case <isim listesi> [(<tür>)]
```

Örneğin:

```
enum Day {
    case Sunday
    case Monday
    case Tuesday
    case Wednesday
    case Thursday
    case Friday
    case Saturday
}
```

Aynı bldirim şöyle de yapılabilirdi:

```
enum Day {
    case Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
}
```

Ancak aynı satıra ';' olmadan birden fazla case yerleştirilemez. Örneğin aşağıdaki bildirim geçerlidir:

```
enum Direction {
    case Up; case Right; case Down; case Left
}
```

Swift'te her enum ayrı bir tür belirtmektedir. Enum sabitlerine enum ismi ve nokta operatörüyle erişilir. Enum sabitlerinin her biri bildirildiği enum türündendir. Aynı türden iki enum birbirlerine atanabilir. Böylece biz bir enum türünden değişkene aynı enum türünün bir enum sabitini doğrudan atayabiliriz. C#, Java gibi dillerde olduğu gibi enum türleriyle tamsayı türleri arasında ve farklı iki enum türü arasında otomatik dönüştürme yoktur. Örneğin:

```
var d: Day;
d = Day.Friday           // geçerli
print(d)                 // Friday
d = 1                    // error!
```

Bir enum türünü print fonksiyonuyla yazdırdığınızda enum'un sabit isminin yazdırıldığına dikkat ediniz.

Enum sabitleri arka planda derleyici tarafından sayısal bir biçimde işleme sokulmaktadır. Enum türleri fonksiyonlara parametre yoluyla aktarılabilir. Örneğin:

Örneğin:

```
enum Day {
    case Sunday
    case Monday
    case Tuesday
    case Wednesday
    case Thursday
    case Friday
    case Saturday
}

func foo(_ day: Day)
{
    print(day)
}

foo(Day.Thursday)
foo(Day.Saturday)
```

Eğer enum sabitleri aynı türden bir enum'a atanacaksa ya da == ve != operatörleriyle aynı türden bir enum ile karşılaştırılacaksa bu durumda enum sabiti enum ismi hiç belirtilmeden nokta operatörüyle kullanılabilir. Örneğin:

```
enum Day {
    case Sunday
    case Monday
    case Tuesday
    case Wednesday
    case Thursday
    case Friday
    case Saturday
}

func foo(_ day: Day)
{
    print(day)
}
```

```

}

var day: Day

day = .Friday           // geçerli
print(day)

foo(.Saturday)          // geçerli

if day == .Friday {     // geçerli
    print("evet")
}
else {
    print("hayır")
}

```

Fakat örneğin:

```

var d = .Sunday         // error!

```

enum türleri switch işlemine sokulabilir. Örneğin:

```

enum Day {
    case Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
}

var day = Day.Wednesday

switch day {
    case .Sunday:
        print("Pazar")
    case .Monday:
        print("Pazartesi")
    case .Tuesday:
        print("Salı")
    case .Wednesday:
        print("Çarşamba")
    case .Thursday:
        print("Perşembe")
    case .Friday:
        print("Cuma")
    case .Saturday:
        print("Cumartesi")
}

```

Burada switch içerisinde bütün seçeneklerin ele alındığına dikkat ediniz. Dolayısıyla buradaki switch deyimi default kısma gereksinim duymamaktadır.

Swift'te de enum türlerinin "temel değer türleri (raw value type)" söz konusu olabilmektedir. Enum türünün temel değer türü tıpkı C#'ta olduğu gibi enum isminden sonra ':' atomu ile belirtilir. Enum türünün temel değer türleri tamsayı türlerinden biri, gerçek sayı türlerinden biri, String ya da Character türü olabilir. (Bir türün enum türünün temel değer türü olabilmesi için onun Equatable protokolünü desteklemesi gerekir. Bunun bazı ayrıntıları vardır. Protokoller konusunda ele alınacaktır.) Sınıflar ya da yapılar enum türlerinin temel değer türleri olamazlar. Örneğin:

```
enum Color : Int {           // Color enum türünün temel değer türü Int
    case Red, Green, Blue
}
```

Eğer enum türüne temel değer türü iliştilmişse artık tüm enum sabitlerinin bu temel değer türünden değer almış olması gerekir. Biz istersek her enum sabitine '=' ile değer atayabiliriz. Bir enum sabitine değer atamamışsak onun değeri onun solundakinin bir fazla değeri olarak atanmaktadır. Eğer ilk elemana değer atanmamışsa ona 0 değerinin atandığı varsayılmaktadır. Örneğin:

```
enum Color : Int {
    case Red = 1, Green, Blue
}
```

Burada Color.Red = 1, Color.Green = 2, Color.Blue = 3 olacaktır. Örneğin:

```
enum Color : Int {           // Temel değer türü Int
    case Red, Green, Blue
}

print(Color.Red.rawValue)    // 0
print(Color.Green.rawValue)  // 1
print(Color.Blue.rawValue)   // 2
```

Ancak diğer değerlerin önceki değerlerin bir fazlasını izlemesi için listede sola doğru ilk kez değer verilen enum sabitine tamsayı bir değerin verilmiş olması gerekir. Örneğin:

```
enum Direction : Double {
    case North = 1.5, East, South = 4, West = 3    // error 1.5 tamsayı olmadığı için East değer
    almamış durumda
}
```

Fakat örneğin:

```
enum Direction : Double {
    case North = 1, East, South = 4.5, West = 3    // geçerli
}
```

Enum değişkeni içerisindeki değer rawValue property'si ile elde edilebilir. Enum'un temel değer türü hangi türse rawValue property'si de bize o türden bir değer verir.

```
enum Color : Int {
    case Red = 1, Green, Blue
}

var c = Color.Blue
print(c.rawValue)    // 3
print(Color.Blue.rawValue) // 3
```

Eğer enum türünün temel değer türü belirtilmemişse biz enum sabitlerine değer atayamayız ve enum türünün rawValue property'sini de kullanamayız. Örneğin:

```
enum Color {
    case Red = 1, Green, Blue    // error! Color enum'unun rawValue property'si yok!
```

```
}
```

Ya da örneğin:

```
enum Color {  
    case Red, Green, Blue  
}  
  
var c = Color.Blue  
print(c.rawValue) // error! Color enum'unun rawValue property'si yok!
```

Swift'teki enum türleri metotlar da içerebilir. Hatta enum türlerinin başlangıç metotları (init metotları) söz konusu olabilir. Enum metotları içerisinde self anahtar sözcüğü yine enum değişkeninin kendisini temsil etmektedir. Örneğin:

```
enum Direction {  
    case North, East, South, West  
  
    init()  
    {  
        self = .South  
    }  
  
    init(d: Direction)  
    {  
        self = d  
    }  
  
    func disp()  
    {  
        print(self)  
    }  
}  
  
var d: Direction = Direction()  
  
d.disp()
```

Eğer enum türüne temel bir değer türü iliştilmişse bu durumda ister biz enum için bir init metodu yazmış olalım istersek yazmamış olalım her zaman derleyici bu enum türü için init(rawValue:) biçiminde bir init metodu yazmaktadır. Ancak bu init metodu bize enum türü E olmak üzere E? biçiminde seçeneksel bir enum değeri vermektedir. Çünkü rawValue ile belirtilen değere ilişkin bir enum sabiti yoksa nil değer oluşabilmektedir. Örneğin:

```
enum Day : Int {  
    case Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday  
}  
  
let d = Day(rawValue: 2)  
print(type(of: d)) // Optional<Day>  
  
if let dval = d {  
    print(dval) // Tuesday  
}  
else {
```

```
    print("Nil")
}
```

Eğer enum türlerinin init metotlarında içeriyorsa bu metot içerisinde self ile enum nesnesine değer verilmesi zorunludur.

Swift'te temel değer türüne sahip olmayan enum'ların elemanlarına ayrı birer değer de iliştilirilebilir. İliştirilecek değerlerin türleri enum sabit isminden sonra parantezler içerisinde belirtilir. Örneğin:

```
enum Fruit {
    case Apple(Int)
    case Banana(Double)
    case Cherry
}

let f1 = Fruit.Apple(10)
let f2 = Fruit.Banana(12.3)
let f3 = Fruit.Cherry

print(f1, f2, f3)
```

Burada Fruit enum türünün Apple elemanı Int türüyle, Banana elemanı Double türüyle ilişkilidir. Cherry elemanı da hiçbir türle ilişkili değildir. Tabii bir enum türünden değişken enum'un herhangi bir elemanını tutabilir. Örneğin:

```
enum Fruit {
    case Apple(Int)
    case Banana(Double)
    case Cherry
}

var f: Fruit = .Banana(10)
print(f)
f = .Apple(10)           // geçerli
print(f)
```

Burada bir noktaya dikkat çekmek istiyoruz. Enum türüne bir temel değer türü iliştilirilmişse artık biz elemanlara ayrıca başka değerler atayamayız. Örneğin:

```
enum Fruit : Int {
    case Apple(Int)           // error!
    case Banana(Double)       // error!
    case Cherry
}
```

Tabii enum sabitine bir demetle de değer iliştilirilebilir. Örneğin:

```
enum Fruit {
    case Apple(Int, Int)
    case Banana(Double, Double)
    case Cherry
}

let f = Fruit.Apple(100, 200)
```



```
print(f)
```

Görüldüğü gibi Swift'te enum sabitlerinin her biri farklı türlerden değerleri tutabilir. Pekiyi bu durumda enum türünden bir değişken bellekte kaç byte yer kaplayacaktır? Örneğin:

```
enum Info {
    case Name(String)
    case IdentityNo(Int)
}

var info: Info      // info ne kadar yer kaplayacak
```

İşte enum türlerini C/C++'taki birliklere (unions) benzetebiliriz. Tipik olarak Swift derleyicisi enum türünden bir değişken için o enum'un en uzun elemanın türü kadar yer ayırmaktadır.

Enum türlerindeki rawValue property'si eğer enum türü temel değer türüne sahipse bulunmaktadır. Dolayısıyla biz enum elemanlarına iliştilmiş olan değerleri rawValue ile elde edemeyiz.

Şimdi bir enum sabitine tür bilgisi atayalım ve ona aşağıdaki gibi bir değer vermiş olalım:

```
var f: Fruit = .Banana(12.4)
```

Burada f'in içerisinde Banana vardır ve ona 12.4 değeri iliştilmiştir. Swift tasarımına göre programcı için asıl önemli olan f'te ne olduğudur. O da örneğimizde Banana'dır. Enum elemanlarına iliştilen değerler switch deyimi içerisinde ya da if deyimi ile elde edilebilmektedir. Bu değerleri elde etmenin diğer bir yolu da "reflection" uygulamaktır. Örneğin:

```
enum Fruit {
    case Apple(Int)
    case Banana(Double)
    case Cherry
}

var f: Fruit = .Banana(12.4)

switch f {
    case .Apple(let a):
        print("Apple: \(a)")
    case .Banana(let b):
        print("Banana: \(b)")
    case .Cherry:
        print("Cherry")
}
```

Tabii yukarıda da belirttiğimiz gibi aslında enum sabitine iliştilen değer ikincil önemdedir. Biz switch deyiminde deyiminde bu değerler olmadan sabit üzerinde karşılaştırma yapabiliriz. Örneğin:

```
enum Fruit {
    case Apple(Int)
    case Banana(Double)
    case Cherry
}
```

```

var f: Fruit = .Banana(12.4)

switch f {
    case .Apple(let a):
        print("Apple: \(a)")
    case .Banana(let b):
        print("Banana: \(b)")
    case .Cherry:
        print("Cherry")
}

```

Enum elemanına iliştilen değer if deyiminde case anahtar anahtar sözcüğü kullanılarak da elde edilebilir. Örneğin:

```

enum Fruit {
    case Apple(Int)
    case Banana(Double)
    case Cherry
}

let f = Fruit.Banana(12.4)

if case .Banana(let x) = f {
    print(x)
}

```

Ya da hiç iliştilen değeri almadan da yine case anahtar sözcüğü kullanılarak karşılaştırma yapılabilir. Örneğin:

```

if case .Banana = f {
    print("Banana")
}

```

Eğer enum elemanlarına demetsel bir bilgi iliştilmişse o değerler yine switch ya da if case deyiminde elde edilebilir. Örneğin:

```

enum Fruit {
    case Apple(Int, Int, Int)
    case Banana(Double)
    case Cherry
}

var f: Fruit = .Apple(10, 20, 30)

switch f {
    case .Apple(let a, let b, let c):
        print("Apple: \(a), \(b), \(c)")
    case .Banana(let b):
        print("Banana: \(b)")
    case .Cherry:
        print("Cherry")
}

```

Demet söz konusu olduğunda case anahtar sözcüğünden sonra tek bir let ya da var anahtar sözcüğü ile demetin tüm elemanları için let ya da var belirlemesi yapılabilmektedir. Örneğin:

```
enum Fruit {
    case Apple(Int, Int, Int)
    case Banana(Double)
    case Cherry
}

var f: Fruit = .Apple(10, 20, 30)

switch f {
case let .Apple(a, b, c):
    print("Apple: \(a), \(b), \(c)")
case .Banana(let b):
    print("Banana: \(b)")
case .Cherry:
    print("Cherry")
}
```

Fonksiyonlar Türünden Değişkenlerin Bildirilmesi

Fonksiyonlar da bellekte ardışıl byte toplulukları biçiminde bulunmaktadır. Onların da adresleri vardır. Örneğin aşağıda iki parametresinin toplamına geri dönen add isimli bir fonksiyonun IA32'deki sembolik makine dili karşılığını görüyorsunuz:

```
add:
    push    ebp
    mov     ebp, esp
    mov     eax, [ebp + 8]
    add     eax, [ebp + 12]
    ret
```

Bir fonksiyonun yalnızca ismi (parantezler olmadan) onun bellekteki adresi anlamına gelir. Örneğin:

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

Burada add ismi bu fonksiyonun bellekteki başlangıç adresi anlamına gelir. Fonksiyonu çağırırken kullandığımız parantezler ise “o adresteki fonksiyonu çağır” anlamına gelmektedir. Örneğin:

```
result = add(10, 20)
```

Burada add adresinden başlayan fonksiyon çağırılmış ve onun geri dönüş değeri result değişkenine atanmıştır.

Swift'te bir fonksiyonu tutabilecek bir değişken bildirilebilir. Bu açıdan fonksiyon türünden değişkenler de birinci sınıf vatandaş (first class citizen) statüsündedir. Fonksiyonu tutabilecek değişkenlerin türleri aşağıdaki gibi oluşturulmaktadır:

```
([tür listesi]) -> <geri dönüş değerinin türü>
```

Geri dönüş değeri olmayan fonksiyonların türleri ise şöyle belirtilir:

```
([tür listesi] -> ())
```

ya da:

```
([tür listesi]) -> Void
```

Aslında Void sözcüğü zaten aşağıdaki gibi typealias yapılmıştır:

```
typealias Void = ()
```

Yani Void demekle () demek tamamen aynı anlamdadır (typealias'lar sonraki konularda ele alınmaktadır).

Fonksiyon türünden bir değişkene ancak parametre türleri ve geri dönüş değeri aynı olan fonksiyonların ya da metodların adresleri atanabilir.

Örneğin:

```
func square(a: Int) -> Int
{
    return a * a
}

var f: (Int) -> Int
var result: Int

f = square                // geçerli, foo'nun parametresi ve geri dönüş değeri Int
result = f(10)            // foo çağrılır, geri dönüş değeri elde edilir
print(result)
```

f bir fonksiyonu tutan değişken olsun. f değişkeninin tuttuğu fonksiyon yine f(...) ifadesi ile çağrılmaktadır.

Örneğin:

```
result = f(10)
print(result)           // 100
```

Bir fonksiyon türünden değişken her türlü fonksiyonun adresini tutamaz. Ancak geri dönüş değeri ve parametreleri belirli biçimde olan fonksiyonların adreslerini tutabilir. Örneğin:

```
var t: (Int, Int) -> Double
```

Burada t yalnızca geri dönüş değeri Double türünden, parametreleri de Int, Int türden olan fonksiyonların adreslerini tutabilir.

Görüldüğü gibi Swift'te fonksiyon türleri kategori olarak referans türlerine ilişkindir. Yani bir fonksiyon türünden değişken fonksiyonun kendisini değil onun adresini tutmaktadır. Yukarıdaki örnekte f fonksiyonu foo fonksiyonunun adresini tutmaktadır. Bundan sonra "bir değişkenin bir fonksiyonu tuttuğundan" söz edildiğinde onun adresini tuttuğu anlaşılmalıdır.

Aynı türden iki fonksiyon değişkeni birbirine atandığında aslında bunların içerisindeki adresler birbirlerine atanmaktadır.

```

func foo(a: Int) -> Int
{
    return a * a
}

var f: (Int) -> Int
var g: (Int) -> Int
var result: Int

f = foo                // geçerli
result = f(10)
print(result)          // 100

g = f                  // geçerli
result = g(10)
print(result)          // 100

```

Böyle işlemlerin C ve C++'ta fonksiyon göstericileri (pointer to functions) ile C#'ta da delegeler (delegates) ile yapıldığını anımsayınız.

Her elemanı bir fonksiyon olan (yani fonksiyonun adresini tutan) bir dizi söz konusu olabilir. Örneğin:

```

func foo(a: Int) -> Int
{
    return a * a
}

func bar(a: Int) -> Int
{
    return a * a * a
}

func tar(a: Int) -> Int
{
    return a * a * a * a
}

let fs: [(Int)->Int] = [foo, bar, tar]

print(fs[2](2))

for f in fs {
    print(f(2))
}

```

Bir fonksiyon değişkeni yapı ya da sınıfların içerisindeki metotların da adreslerini tutabilir. Bu durumda bizim ilgili değişkene sınıf ya da yapı değişkeni ile metot ismini nokta operatörü ile birleştirerek vermemiz gerekir. (Bu kısmın C#'taki delegelere çok benzediğine dikkat ediniz). Örneğin:

```

struct Sample {
    var a: Int

    init(a: Int)
    {
        self.a = a
    }
}

```

```

    }

    func disp()
    {
        print(a)
    }
}

var t: Sample = Sample(a: 10)
var f: () -> Void = t.disp

f()    // t.disp() ile aynı anlamda

```

Örneğin:

```

struct Sample {
    var a: Int

    init(a: Int)
    {
        self.a = a
    }

    func disp(str: String)
    {
        print("\(str): \((a)")
    }
}

var t: Sample = Sample(a: 10)
var f: (String) -> Void = t.disp

f("Value")    // t.disp("Value") ile aynı anlamda

```

Fonksiyon değişkenlerinde etiket isimlerinin belirtilmediğine dikkat ediniz. Yani fonksiyon değişkeni parametre türleri ve geri dönüş değeri uymak koşuluyla farklı etiketlere sahip fonksiyonları ya da metotları tutabilir.

Benzer biçimde statik metotlar da fonksiyon değişkenlerine sınıf ismi belirtilerek atanmalıdır:

```

struct Sample {
    static func foo()
    {
        print("foo")
    }
}

var f: () -> Void = Sample.foo

f()    // Sample.foo() ile eşdeğer

```

Bir fonksiyonun parametre değişkeni bir fonksiyon türünden olabilir. Örneğin:

```

func foo(a: Int, f: (Int)->Int)
{
    let result = f(a)
    print(result)
}

```

```

}

func square(a: Int) -> Int
{
    return a * a
}

foo(a: 10, f: square)

```

Örneğin:

```

func forEachString(_ strs: [String], _ f: (String) -> Void)
{
    for str in strs {
        f(str)
    }
}

func disp(_ str: String)
{
    print(str)
}

let names = ["Ali", "Veli", "Selami", "Ayşe", "Fatma"]
forEachString(names, disp)

```

Swift'in standart kütüphanesindeki bazı fonksiyonlar ve metotlar da parametre olarak fonksiyon türünden değişkenlere sahip olabilmektedir. Örneğin Array yapısının sort isimli iki metodu vardır. Birinci sort metodu parametresizdir. Eğer söz konusu dizi türü Comparable protokolünü destekliyorsa bu parametresiz sort metodu küçükten büyüğe sıralama yapmaktadır. Örneğin:

```

var names = ["Ali", "Veli", "Selami", "Ayşe", "Fatma"]

names.sort()
for name in names {
    print(name, terminator: " ")
}
print()

```

İkinci sort fonksiyonu bizden bir karşılaştırma fonksiyonu alır. Sıraya dizme sırasında dizinin iki elemanını bu karşılaştırma fonksiyonuna sokarak duruma göre yer değiştirme uygular. Karşılaştırma fonksiyonunun parametrik yapısı şöyledir:

```

cmp(a: T, b: T) -> Bool

```

Burada T dizinin türünü belirtiyor. Eğer biz diziyi küçükten büyüğe sıraya dizeceksek ilk parametre ikinci parametreden küçükse true değerine, değilse false değerine geri dönmeliyiz. Örneğin:

```

var names = ["Ali", "Veli", "Selami", "Ayşe", "Fatma"]

func mycmp(_ a: String, _ b: String) -> Bool
{
    return a < b
}

```

```

names.sort(by: mycmp)
for name in names {
    print(name, terminator: " ")
}
print()

```

Örneğin:

```

struct Person {
    let name: String
    let no: Int

    func disp()
    {
        print("\(name), \(no)")
    }
}

func name(_ a: Person, _ b: Person) -> Bool
{
    return a.name < b.name
}

func no(_ a: Person, _ b: Person) -> Bool
{
    return a.no < b.no
}

var persons = [
    Person(name: "Kaan Aslan", no: 123),
    Person(name: "Selami Karakelle", no: 513),
    Person(name: "Ali Serçe", no: 317),
    Person(name: "Necati Ergin", no: 999),
    Person(name: "Lokman Köse", no: 423)
]

persons.sort(by: name)

for person in persons {
    person.disp()
}

print("-----")

persons.sort(by: no)

for person in persons {
    person.disp()
}

```

Array<T> yapısının sort dışında ayrıca sorted isimli metotları da vardır. Bu metotlar asıl diziyi sıraya dizmezler sıraya dizilmiş yeni bir dizi verirler. Böylece asıl dizi let de olabilir. Örneğin:

```

let names = ["Ali", "Veli", "Selami", "Ayşe", "Fatma"]
var sortedNames: [String]

```



```
sortedNames = names.sorted()
for name in sortedNames {
    print(name, terminator: " ")
}
print()
```

Bir fonksiyonunun geri dönüş değeri de bir fonksiyon türünden olabilir. Yani fonksiyon bize geri dönüş değeri olarak bir fonksiyon verebilir. Örneğin:

```
func inc(a: Int) -> Int
{
    return a + 1
}

func dec(a: Int) -> Int
{
    return a - 1
}

func which(_ a: Int) -> (Int) -> Int
{
    return a > 0 ? inc : dec
}

var f: (Int) -> Int
f = which(10)

print(f(100))           // 101
print(which(10)(100))   // 101
```

Closure'lar

Closure'lar C++, C# ve Java'daki lambda ifadelerinin Swift'teki karşılığıdır. Closure bir fonksiyonun ya da metodun bir ifade içerisinde bildirilip kullanılması anlamına gelir. Closure bildiriminin genel biçimi şöyledir:

```
{ ([parametre bildirimi]) [-> <geri dönüş değerinin türü>] in [deyimler] }
```

Closure'lar fonksiyon türlerindendir. Bir closure bildirimi sanki fonksiyonu bildirip onun adresini kullanmak gibidir. Örneğin:

```
var f: (Int) -> Int

f = {(a: Int) -> Int in return a * a}
```

işlemi ileride ele alınacak bazı ayrıntılar dışında aşağıdaki ile eşdeğerdir:

```
var f: (Int) -> Int

func foo(a: Int) -> Int
{
    return a * a
}
```

```
f = foo
```

Örneğin:

```
var f: (Int)->Int
```

```
f = {(a: Int) -> Int in return a * a}  
print(f(10))
```

Bir fonksiyonun parametresi fonksiyon türündense biz onu çağırırken argüman olarak closure verebiliriz. Örneğin:

```
func foo(val: Int, f: (Int) -> Int)  
{  
    print(f(val))  
}
```

```
foo(val: 10, f: {(a: Int)->Int in return a * a})
```

Burada foo fonksiyonunun birinci parametresi geri dönüş değeri Int ve parametresi Int türden olan bir fonksiyon türündendir. (Yani parametre değişkeni fonksiyonun adresini alır). Fonksiyon çağrılırken doğrudan aynı türden closure verilmiştir. Örneğin:

```
var names = ["Ali", "Veli", "Selami", "Ayşe", "Fatma"]  
  
names.sort(by: {(a: String, b: String)->Bool in return a < b})  
for name in names {  
    print(name, terminator: " ")  
}  
print()
```

in anahtar sözcüğünden sonra closure içerisine istenildiği kadar deyim yerleştirilebilir. Örneğin:

```
func foo(val: Int, f: (Int) -> Int)  
{  
    print(f(val))  
}
```

```
foo(val: 10, f: {  
    (a: Int)->Int in  
    print("Ok")  
    if a > 0 {  
        return a * a * a  
    }  
    else {  
        return a * a  
    }  
})
```

Closure parametreleri yine default olarak let biçimdedir.

Eğer closure doğrudan bir fonksiyon türünden değişkene atanıyorsa closure'larda fonksiyon parametre türleri hiç belirtilmeyebilir. Bu durumda closure nasıl bir fonksiyon türünden değişkene atanmışsa parametrelerin de o türden olduğu kabul edilir. Örneğin:

```
var f: (Int, Int) -> Int

f = {(a, b) -> Int in return a + b }    // a ve b Int türden
print(f(10, 20))
```

Eğer parametre türleri belirtilmeyecekse parantezler de kullanılmayabilir. Örneğin:

```
var f: (Int, Int) -> Int

f = {a, b -> Int in return a + b }      // a ve b Int türden
print(f(10, 20))
```

Benzer biçimde closure'ların atandığı fonksiyon değişkeninin hangi geri dönüş değerine sahip bir fonksiyon türünden olduğu bilindiği için closure'ların geri dönüş değerlerinin türünün de belirtilmesine gerek yoktur. Örneğin:

```
var f: (Int, Int) -> Int

f = {a, b in return a + b }    // a ve b Int türden geri dönüş değeri de Int türden
print(f(10, 20))
```

Örneğin:

```
var names = ["Ali", "Veli", "Selami", "Ayşe", "Fatma"]

names.sort(by: {a, b in return a < b})
for name in names {
    print(name, terminator: " ")
}
print()
```

Eğer in anahtar sözcüğünden sonra tek bir ifade varsa bu durumda return anahtar sözcüğünün de kullanılmasına gerek yoktur. Örneğin:

```
var f: (Int, Int) -> Int

f = {a, b in a * b }    // return anahtar sözcüğüne gerek yok
print(f(10, 20))
```

Örneğin:

```
var names = ["Ali", "Veli", "Selami", "Ayşe", "Fatma"]

names.sort(by: {a, b in a < b})    // return anahtar sözcüğüne gerek yok
for name in names {
    print(name, terminator: " ")
}
print()
```

Tabii closure'ın atandığı fonksiyon değişkeninin ilişkin olduğu fonksiyon türünün geri dönüş değeri Void olabilir. Bu durumda in anahtar sözcüğünün yanındaki ifade geri dönüş değeri anlamına gelmez. Örneğin:

```
var f: (Int, Int) -> Void
```

```
f = {a, b in print("\(a), \b)} // burada return yok
f(10, 20)
```

Nihayet closure’larda parametre değişken isimleri bile belirtilmeyebilir. Bu durumda parametre değişkenleri sırasıyla \$0, \$1, \$2, ... isimleriyle kullanılabilir. Parametre değişken isimleri belirtilmemişse geri dönüş değerinin türü de in anahtar sözcüğü de artık kullanılamaz. Böylelikle closure’lar çok kısa biçimde ifade edilebilmektedir. Örneğin:

```
var f: (Int, Int) -> Int

f = { $0 + $1 } // iki parametresinin toplamına geri dönüyor
print(f(10, 20))
```

Tabii yine istersek return anahtar sözcüğünü kullanabiliriz. Örneğin:

```
var f: (Int, Int) -> Int

f = {return $0 + $1}
print(f(10, 20))
```

Örneğin:

```
var f: () -> ()

f = { print("test") }
f()
```

Örneğin:

```
var names = ["Ali", "Veli", "Selami", "Ayşe", "Fatma"]

names.sort(by: {$0 < $1})
for name in names {
    print(name, terminator: " ")
}
print()
```

Eğer fonksiyon değişkeni o anda let ya da var ile bildiriliyorsa bu durumda derleyicinin closure parametrelerinin türlerini belirleyebiliyor olması gerekir. Örneğin:

```
let f = {(a: Int, b: Int) -> Int in return a + b} // geçerli
print(f(10, 20))
```

Örneğin:

```
let f = {(a: Int, b: Int) -> Int in return a + b} // geçerli
print(f(10, 20))
```

Fakat örneğin:

```
let f = {print("Test")}
print(type(of: f)) // () -> ()
```

Örneğin:

```
func foo(a: Int) -> Int
{
    return a * a
}

let f = {foo(a: 100)}
print(type(of: f))    // () -> Int
```

Örneğin:

```
let f = {(10, 20)}
print(type(of: f))    // () -> (Int, Int)
```

let ya da var bildirimlerinde eğer closure'da birden fazla ifade varsa ve return kullanılmamışsa geri dönüş değeri her zaman Void kabul edilir. Örneğin:

```
func foo(a: Int) -> Int
{
    return a * a
}

let f = {print("Test"); foo(a: 100)}
print(type(of: f))    // () -> ()
```

Swift'te eğer bir fonksiyon ya da bir metodun son parametresi bir fonksiyon türündense ve biz o fonksiyonu ya da metodu son argümanı closure olacak biçimde çağırıcaksak bu durumda closure içeriğini parametre parantezinden sonra açılan blok içerisinde de yazabiliriz. Örneğin:

```
func foo(a: Int, f: (Int) -> Int)
{
    print(f(a))
}
```

Burada foo fonksiyonunun birinci parametresi Int türden, ikinci parametresi ise parametresi Int ve geri dönüş değeri Int türden olan fonksiyon türündendir. Bu fonksiyonu normal olarak şöyle çağırabiliriz:

```
foo(10, f: {(a: Int) -> Int in return a * a})
```

Burada foo çağrılırken ikinci parametresi closure olarak girilmiştir. İşte bu çağrıyı pratik olarak şöyle de yapabiliydik:

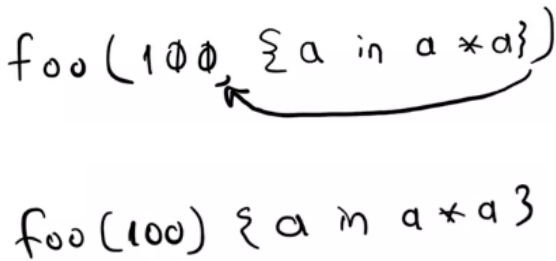
```
foo(10) {
    (a: Int) -> Int in return a * a
}
```

Bu ifadenin bir fonksiyon bildirimi anlamına gelmediğine fonksiyon çağırma anlamına geldiğine dikkat ediniz. Ya da örneğin:

```
foo(10) {
```

```
$0 * $0  
}
```

Bu sentaksın oluşturulmasına dikkat ediniz. Sentakta adeta son çağrı parantezi ',' atomunun yerine kaydırılmış gibidir:



Bir closure dıştaki bloğun değişkenlerini kullanabilir. Closure'lar bu bakımdan iç içe fonksiyonlara benzetilebilir. Örneğin:

```
var x = 10  
  
func foo()  
{  
    let i = 20  
  
    let f: () -> Int = {() -> Int in return i * x }    // let f = { i * x }  
  
    print(f())  
}  
  
foo()    // 200
```

İçteki fonksiyonların ya da closure'ların dış bloktaki değişkenleri kullanmasına Swift terminolojisinde (C++'ta da aynı terim kullanılıyor) "capturing" denilmektedir. Capture işlemini derleyici arka planda etkin olarak optimize etmektedir. İç bloktaki closure dış bloktaki değişkenleri değiştirebilir. Örneğin:

```
func foo()  
{  
    var i = 20  
  
    let f: () -> Void = { i *= 2 }  
  
    print(i)    // 20  
    f()  
    print(i)    // 40  
}  
  
foo()  
foo()
```

Bilindiği gibi yerel değişkenler akış fonksiyondan çıktığında otomatik olarak yok edilmektedir. Ancak bir fonksiyon iç bir fonksiyonla ya da closure ile geri dönebilir. Bu durumda bu iç fonksiyon ya da closure dış fonksiyonun değişkenlerini kullanıyorsa bunlar derleyici tarafından kalıcı hale getirilmektedir. Örneğin:

```

func foo() -> () -> Int
{
    var i = 20

    let f = { () -> Int in i *= 2; return i }

    return f
}

var result: Int

var f = foo()
result = f()
print(result)           // 40
result = f()
print(result)           // 80

var g = foo()
result = g()
print(result)           // 40
result = g()
print(result)           // 80

```

Burada foo fonksiyonunun çalışması bittiğinde capture edilmiş i yok edilmemektedir. Tabii her foo çağırımı yeni bir i'nin yaratılmasına yol açar.

Bir fonksiyon ya da metot bir fonksiyon türünden parametre değişkenine sahipse o parametreyi dışarda bir yere aktarırsa derleyici optimizasyonunu mümkün kılabilmek için parametre özelliği ile özniteliklendirilebilir. Örneğin:

```

var a: [() -> ()] = []

func foo(f: () -> ())
{
    a.append(f)           // error
}

```

Burada çağrı error ile sonuçlanır. Çünkü parametre alınan fonksiyon dışarıda başka bir yere aktarılmıştır. Eğer buradaki error'ün ortadan kaldırılması isteniyorsa @escaping biçiminde özniteliklendirmenin yapılması gerekmektedir. Örneğin:

```

var a: [() -> ()] = []

func foo(f: @escaping () -> ())
{
    a.append(f)           // geçerli
}

```

Bir fonksiyon ya da metot () -> T türünden bir fonksiyon parametresine sahipse onu hiç küme parantezleri olmadan çağırabiliriz. Bunun için parametre değişkeninin türünün @autoclosure özneliği ile özniteliklendirilmesi gerekir. Örneğin:

```

func foo(f: @autoclosure () -> Int)
{

```

```

    print(f())
}

foo(f: 10 + 20)           // geçerli

```

Burada autoclosure işlemi için parametredeki fonksiyon türünden değişkenin parametrelerinin olmaması yalnızca geri dönüş değerinin olması olması (aslında geri dönüş değeri de Void olabilir) gerekmektedir.

Örneğin:

```

func foo(a: Int, f: @autoclosure () -> Void, b: Int)
{
    print(a, b)
    f()
}

foo(a: 10, f: print("Ok"), b: 20)           // geçerli

```

@autoclosure özniteliklendirmesi yapılmış bir fonksiyon ya da metodu fonksiyon ya da closure ile çağıramayız. Yalnızca yukarıda belirtildiği gibi pratik sentaksla çağırabiliriz. Örneğin:

```

func foo(f: @autoclosure () -> Int)
{
    print(f())
}

foo(f: 10 + 20)           // geçerli
foo(f: {()-> Int in 10 + 20}) // error!

```

Swift'te eklenen bir closure kuralı da değer listeleridir (capture value list). Bir closure bildirimin hemen başında köşeli parantezler içerisinde üst bloklardaki değişkenler virgül atomlarıyla ayrılmış bir liste yazılabilir. Bu durumda closure içerisinde bu üst bloktaki değişkenlerin kendileri değil onların closure bloğuna girişteki değerleri kullanılır. Örneğin:

```

func foo()
{
    var a, b: Int
    a = 10
    b = 20

    let f = {
        () -> () in print(a + b);
    }

    a = 30
    b = 40

    f()           // ekrana 70 yazılır
}

foo()

```

Burada closure'a a ve b'nin kendisi geçirilmiştir. Yani closure çağrıldığında çağırılma noktasındaki değerleri closure kullanır. Ayrıca dış bloktaki değişkenler let değilse closure onları aynı zamanda değiştirebilmektedir.

Halbuki biz dış bloklardaki değişkenlerden köşeli parantezler içerisinde bir liste oluşturursak bu durumda closure bildirimine girişte o değişkenlerin değerleri kopyalanarak closure'a aktarılır. Ayrıca closure içerisinde artık biz bu değişkenlerin değerlerini değiştiremeyiz. Örneğin:

```
func foo()
{
    var a, b: Int
    a = 10
    b = 20

    let f = {
        [a, b] () -> () in print(a + b);
    }

    a = 30
    b = 40

    f()      // ekrana 30 yazılır
}

foo()
```

Türleri İsimlendirmek (type aliases)

Swift'te de diğer dillerde olduğu gibi bir türe her bakımdan onun yerini tutabilecek alternatif isimler verilebilir. Tür isimlendirmenin genel biçimi şöyledir:

```
typealias <yeni tür ismi> = <tür ismi>
```

Örneğin:

```
typealias I = Int      // I ile Int aynı anlamda

var a: I
a = 100

print(a)
```

Karmaşık tür ifadeleri typealias kullanılarak daha sade ifade edilebilir. Örneğin:

```
typealias Proc = () -> ()

var f: Proc
f = { () -> () in print("ok")}      // f = { print("Ok")}
f()
```

Örneğin:

```
typealias Procs = [() -> ()]

let procs: Procs = [{() -> () in print("one")}, {() -> () in print("two")}]
// let procs: Procs = [{print("one")}, {print("two")} ]

for f in procs {
```

```
f()
}
```

typealias bildirimi ile oluşturulmuş olan değişkenler global ya da yerel faaliyet alanına sahip olabilir. Yani örneğin biz bir fonksiyonun içerisinde bir typealias oluşturduğumuzda bu tür ismini yalnızca o fonksiyonun içerisinde kullanabiliriz. Örneğin:

```
func foo()
{
    typealias I = Int

    let a: I = 10
    print(a)
}

func bar()
{
    let b: I = 20      // error!
    //...
}
```

Sınıflar

Swift'te sınıflarla yapılar bildirim ve kullanım bakımından birbirlerine çok benzemektedir. Sınıf bildiriminin genel biçimi şöyledir:

```
class <sınıf ismi> {
    //...
}
```

Sınıflarla daha önce gördüğümüz yapıların benzerlikler şöyle özetlenebilir:

- Sınıflar da property elemanlara (veri elemanlarına) ve metotlara sahiptir.
- Sınıfların da başlangıç metotları (init metotları) vardır.
- Sınıflarda da elemana erişim yine nokta operatörüyle yapılmaktadır.
- Sınıflar da static ve static olmayan elemanlara sahip olabilirler.
- Sınıflar da subscript elemanlara sahip olabilir.
- Sınıflar da yapılar gibi protokolleri destekleyebilirler.

Ancak Swift'te sınıflar yapılardan daha fazla özelliklere sahiptir ve sınıfların daha geniş bir kullanım alanı vardır.

Sınıflar kategori olarak referans türlerine ilişkindir. Yani bir sınıf türünden bir değişken (referans) bildirdiğimizde o değişken sınıf nesnesinin adresini tutar. Örneğin:

```
struct A
{
    var x: Int
    var y: Int
    //...
}

class B
```

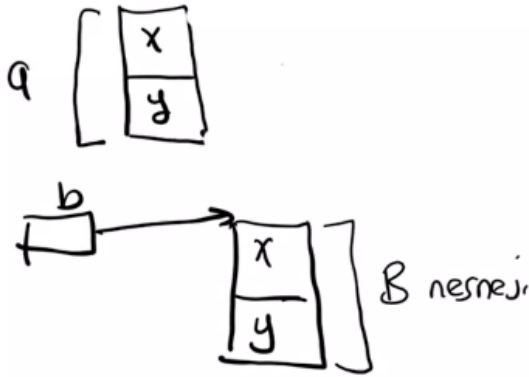
```

{
    var x: Int
    var y: Int
    //...
}

let a = A()
let b = B()

```

Bu örnekte a bir yapı türündendir. a kendi içerisinde bileşik bir nesnedir. Yani x ve y parçalarına sahiptir. Halbuki b bir sınıf türünden olduğu için ve sınıf türleri kategori olarak referans türlerine ilişkin olduğu için b değişkeni yalnızca adres tutan bir değişkendir. Asıl nesne b değişkeninin gösterdiği adreste bulunur.



Sınıflar türünden nesneler yaratıldığında yaratım heap denilen bölgede yapılır ve bu işlemde yaratılan nesnenin adresi elde edilir. Halbuki bir yapı nesnesi yaratıldığında stack'ta geçici bir bileşik nesne yaratılmaktadır. Başka bir deyişle yapı nesneleri stack'te sınıf nesneleri heap'te yaratılırlar. Örneğin:

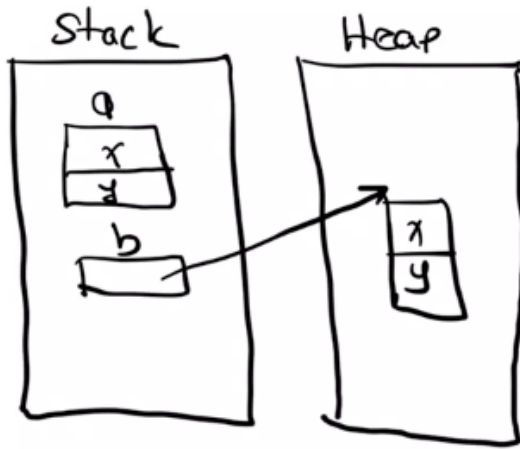
```

func foo()
{
    var a: A
    var b: B

    a = A()
    b = B()
    //...
}

```

Burada a ve b yerel değişken olduğu için stack'tedir. a bileşik bir nesnedir. Bu nesnenin her parçası da dolayısıyla stack'te tutulmaktadır. Halbuki b değişkeni stack'te olsa da onun gösterdiği yerdeki nesne heap'tedir:



Biz kursumuzda sınıf türünden değişkenlere kısaca referans da diyeceğiz. Aynı türden iki sınıf değişkeni birbirlerine atanabilir. Bu durumda aslında onların içerisindeki adresler atanmaktadır. Böylece iki sınıf değişkeni de aynı nesneyi gösterir hale gelir. Örneğin:

```
class Sample {
    var a: Int
    var b: Int

    init(a: Int, b: Int)
    {
        self.a = a
        self.b = b
    }
}

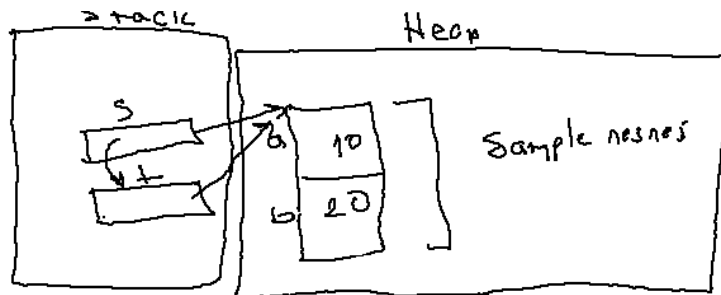
var s, t: Sample

s = Sample(a: 10, b: 20)
t = s

print("s.a = \(s.a), s.b = \(s.b)")    // s.a = 10, s.b = 20
print("t.a = \(t.a), t.b = \(t.b)")    // t.a = 10, t.b = 20

t.a = 30
t.b = 40

print("s.a = \(s.a), s.b = \(s.b)")    // s.a = 30, s.b = 40
print("t.a = \(t.a), t.b = \(t.b)")    // t.a = 30, t.b = 40
```



Aynı türden iki sınıf referansı birbirlerine atandığında onların içerisindeki adreslerin atandığına dikkat ediniz. Dolayısıyla iki referans da aynı nesneyi gösterir duruma gelmektedir. Yukarıdaki işlem yapıyla gerçekleştirilseydi şöyle bir sonuç oluşurdu:

```
struct Sample {
    var a: Int
    var b: Int

    init(a: Int, b: Int)
    {
        self.a = a
        self.b = b
    }
}

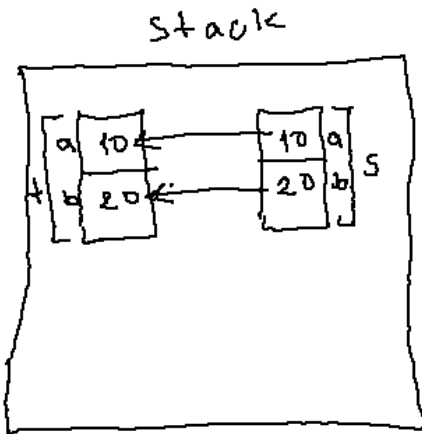
var s, t: Sample

s = Sample(a: 10, b: 20)
t = s

print("s.a = \(s.a), s.b = \(s.b)")    // s.a = 10, s.b = 20
print("t.a = \(t.a), t.b = \(t.b)")    // t.a = 10, t.b = 20

t.a = 30
t.b = 40

print("s.a = \(s.a), s.b = \(s.b)")    // s.a = 10, s.b = 20
print("t.a = \(t.a), t.b = \(t.b)")    // t.a = 30, t.b = 40
```



Sınıf türünden bir değişkenin let olması o değişkenin kendisine başka bir adres atanamayacağı anlamına gelmektedir. O değişkenin gösterdiği sınıf nesnesinin elemanlarına değer atanabilir. Örneğin:

```
class Sample {
    var a: Int = 10
    var b: Int = 20
}

let s = Sample()

s.a = 10    // geçerli
s.b = 20    // geçerli
```

```
s = Sample()    // error
```

Sınıflarda init metotları konusunda küçük bir farklılık vardır. Anımsanacağı gibi yapılarda init metodunun derleyici tarafından yazılma kuralı şöyleydi:

- 1) Programcı yapı için herhangi bir init metodu yazmışsa derleyici programcı için hiçbir init metodu yazmaz.
- 2) Programcı yapı için hiçbir init metodu yazmamışsa derleyici eleman ilkdeğerlemesi yapan init metodunu kendisi yazar. Ayrıca derleyici parametresiz init metodunu da programcı eğer yapının tüm elemanlarına bildirim sırasında ilkdeğer verilmişse kendisi yazmaktadır.

Halbuki sınıflar için derleyicinin init metodunu kendisinin yazması kuralı şöyledir:

- 1) Programcı sınıf için herhangi bir init metodu yazmışsa derleyici programcı için hiçbir init metodu yazmaz.
- 2) Programcı parametresiz init metodunu eğer sınıfın tüm elemanlarına bildirim sırasında ilkdeğer verilmişse yazmaktadır. Ancak sınıflarda eleman ilkdeğerlemesi yapan init metodu derleyici tarafından hiçbir biçimde yazılmamaktadır.

Aradaki farka dikkat ediniz: Sınıflarda derleyici hiçbir biçimde eleman ilkdeğerlemesi yapan init metodunu yazmamaktadır.

Yine sınıflarda da init metotların içerisinde bütün veri elemanlarına değer atanmış olması gerekir. Tabii bildirim sırasında veri elemanına değer atanmışsa bu elemanlara ayrıca init metotları içerisinde yeniden değer atanması gerekmez. Örneğin:

```
class Sample {
    let a: Int = 10
    var b: Int
    var c: Int

    init()    // geçerli tüm veri elemanlarına değer atanmış
    {
        b = 10
        c = 20
    }
}
```

Ne Zaman Yapı Ne Zaman Sınıf Tercih Edilmelidir?

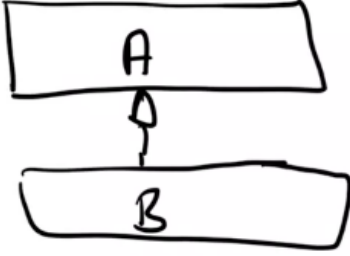
Yapılarla sınıflar pek çok bakımdan birbirlerine benzemektedir. Pekiye programcı ne zaman yapı ne zaman sınıf kullanmayı tercih etmelidir?

- 1) Yapılar stack'te yaratılır. Bunların yaratımları ve yok edimleri sınıf nesnelerine göre çok daha hızlı yapılmaktadır. Bu nedenle yalnızca bir fonksiyon ya da metodun içerisinde kullanılacak bir nesnenin yapı biçiminde bildirilmesi tercih edilebilir.
- 2) Yapı nesneleri yaratıldıkları fonksiyon ya da metot sonlandığında otomatik olarak yok edilirler. Halbuki sınıf nesneleri ilgili fonksiyon ya da metot sonlansa bile heap'te kalmaya devam ederler.

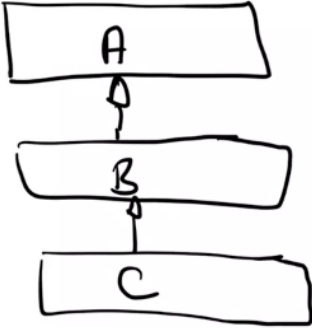
3) Bazı özellikler yalnızca sınıflara özgüdür. Bu tür durumlarda mecburen sınıflar kullanılır. Örneğin yapılardan türetme yapılamaz. Yalnızca sınıflardan türetme yapılabilir.

Türetme İşlemleri

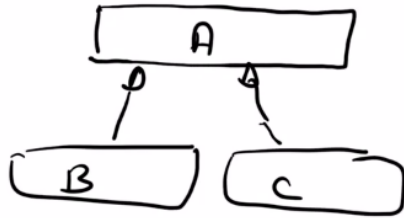
Türetme mevcut bir sınıfı genişletme yani ona birtakım elemanlar ekleme anlamına gelmektedir. Burada asıl sınıfa taban sınıf (base class) eleman eklenerek genişletilmiş olan sınıfa da türemiş sınıf (derived class) denilmektedir. Türetme işlemi UML sınıf diyagramlarında türemiş sınıftan taban sınıfa çekilen içi boş bir ok ile gösterilmektedir. Örneğin:



Burada A bir taban sınıf B de türemiş sınıftır. Yani B hem A gibi hem de B gibi fazlalıklarla kullanılabilir. Türemiş sınıftan yeniden türetme yapılabilir. Örneğin:



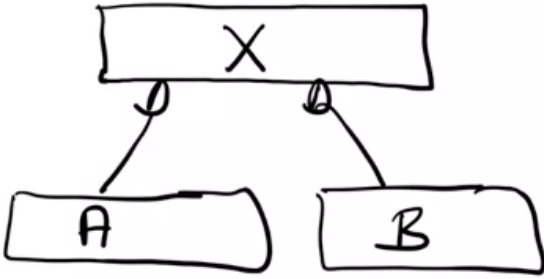
Bir sınıf birden fazla sınıfın taban sınıfı olabilir. Örneğin:



Ancak bir sınıfın birden fazla taban sınıfı olması durumu özel bir durumdur. Buna "çoklu türetme (multiple inheritance)" denilmektedir. Çoklu türetme Swift'te yoktur. Ancak başka bazı dillerde vardır.

Türetmenin en önemli faydası tekrarı engellemektir. Eğer iki sınıfta ortak bazı elemanlar varsa bu elemanların bu iki sınıfa da ayrı ayrı yerleştirilmesi tekrara yol açar. Tekrar ise hem kodu büyütmede hem de kodun bakımını

zorlaştırmaktadır. İşte A ve B'de ortak birtakım elemanlar varsa bunlar bir X sınıfında toplanabilir. A ve B bu X sınıfından türetilir.



Böylelikle bir türetme şemasında yukarıya çıkıldıkça genelleşme aşağıya inildikçe özelleşme görülür. Yani yukarıdaki sınıflar aşağıdakilerin ortak özelliklerini barındırmaktadır.

Anımsanacağı gibi Swift'te değer türleri (yani struct ve enum türleri) türetmeye kapalıdır. Yani bir yapı ya da enum türünden biz bir sınıf, yapı ya da enum türünü, bir sınıf türünden de bir yapı ya da enum türünü türetemeyiz. Ancak sınıflardan sınıflar türetilir.

Swift'te türetme işleminin genel biçimi şöyledir:

```
class <türemiş sınıf ismi> : <taban sınıf ismi> {  
    //...  
}
```

Swift'in türetme sentaksının C++ ve C#'takine benzediğine dikkat ediniz.

Diğer dillerde de olduğu gibi türetme durumunda taban sınıfın elemanları sanki türemiş sınıfın elemanlarıymış gibi işlem görmektedir. Yani biz türemiş sınıf türünden bir referansla hem türemiş sınıfın hem de taban sınıfın elemanlarını kullanabiliriz. Örneğin:

```
class A {  
    var x: Int = 0  
  
    func foo()  
    {  
        print("A.foo")  
    }  
}  
  
class B : A {  
    var y: Int = 0  
  
    func bar()  
    {  
        print("B.bar")  
    }  
}  
  
var b = B();
```



```
b.foo()      // geçerli
b.bar()      // geçerli
print(b.x)   // geçerli
print(b.y)   // geçerli
```

Örneğin:

```
class A {
    var a: Int

    init()
    {
        a = 10
    }

    func dispA()
    {
        print(a)
    }
}

class B : A {
    var b: Int

    override init()
    {
        b = 20
    }

    func dispB()
    {
        print(b)
    }
}

let b = B()

b.dispA()
b.dispB()
```

Burada B sınıfının init metodunda kullanılan override anahtar sözcüğü ileride ele alınacak.

Yine Swift'te de türemiş sınıf türünden bir nesne hem taban sınıf property'lerini i(yani veri alamanlarını) hem de türemiş sınıfın kendi property'lerini (yani veri elemanlarını) içerir. sınıfların ve yapıların metotları nesne içerisinde değildir. Programın kod alanı içerisinde dir. Örneğin:

```

class A {
    var x: Int = 0
    var y: Int = 0
    //~
}

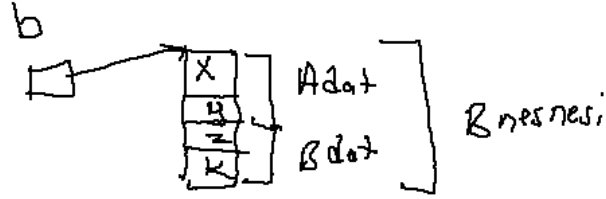
```

```

class B: A {
    var z: Int = 0
    var k: Int = 0
    //~
}

```

```
var b: B = B()
```



Yine Swift'te de türemiş sınıf nesnesinin taban sınıf kısmı ile türemiş sınıf kısmı ardışıl bir blok oluşturur. Nesnenin taban sınıf kısmı bellekte düşük adreste bulunmaktadır. Örneğin:

```

class A {
    var x: Int
    var y: Int

    init()
    {
        x = 10
        y = 20
    }
}

```

```

class B : A {
    var z: Int
    var k: Int

    override init()
    {
        z = 30
        k = 30
    }
}

```

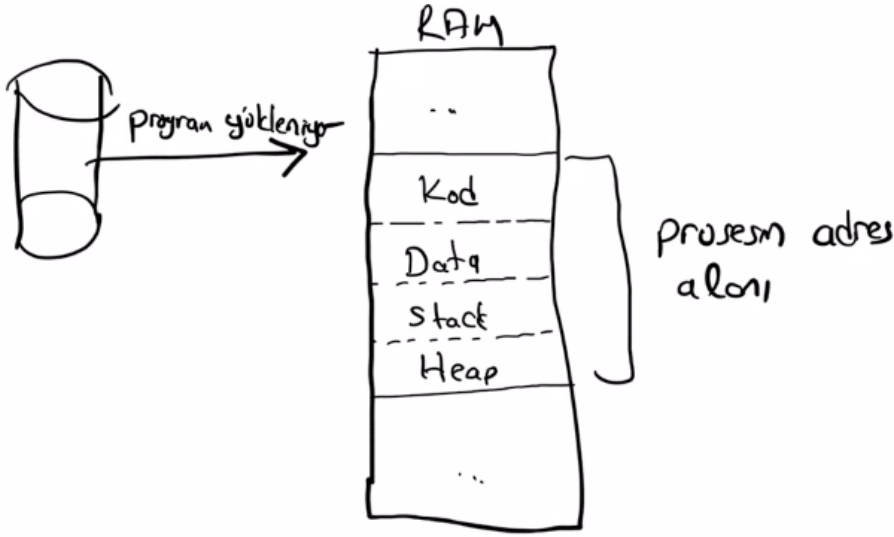
```

let b = B()
print(b.x, b.y, b.z, b.k)

```

Programların Kod, Data, Stack ve Heap Alanları

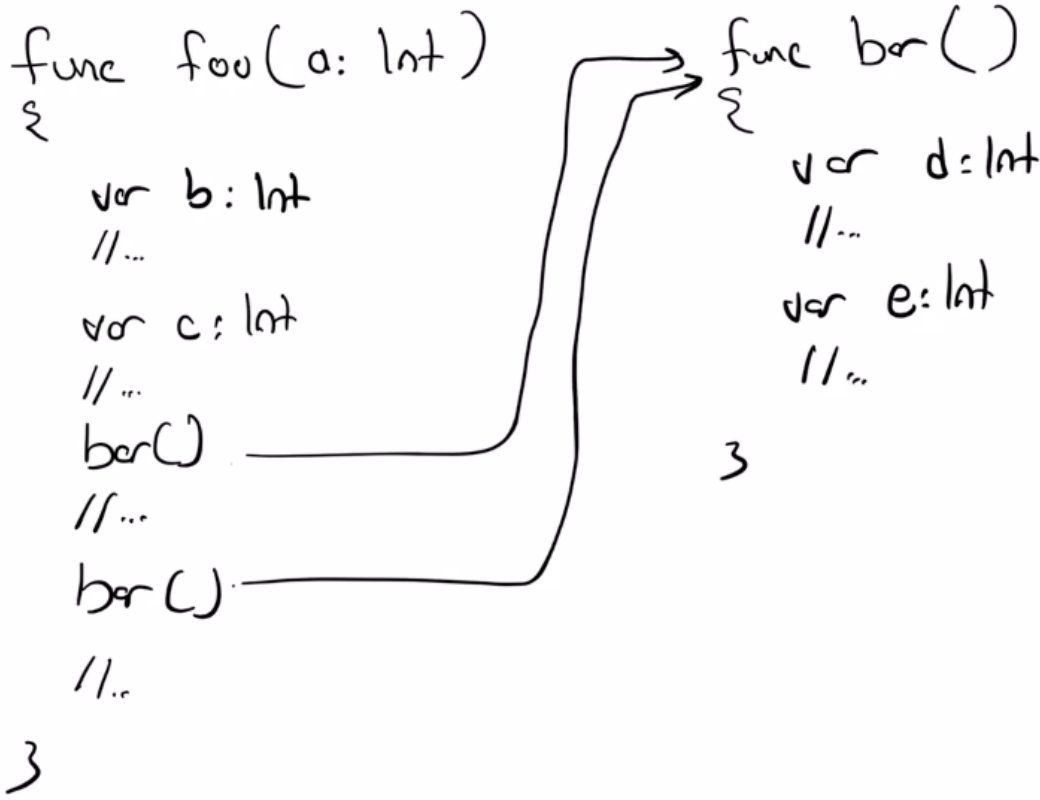
Çalışmakta olan programlara "proses" denilmektedir. Bir program çalışmak üzere işletim sistemi tarafından diskten alınarak belleğe yüklendiğinde oluşturulan proses için bellekte dört alan ayrılmaktadır. Bu bölümlere kod, stack, data ve heap alanları denilmektedir.



Bir programdaki bütün global fonksiyonların ve metotların makine kodları bir araya getirilerek kod bölümünde toplanmaktadır. Yani yapıların ya da sınıfların metotları yaratılan nesnenin içerisinde değildir. Bunlar kod bölümünde bulunurlar. Metotların yapılarla ve sınıflarla ilişkisi mantıksal düzeydedir.

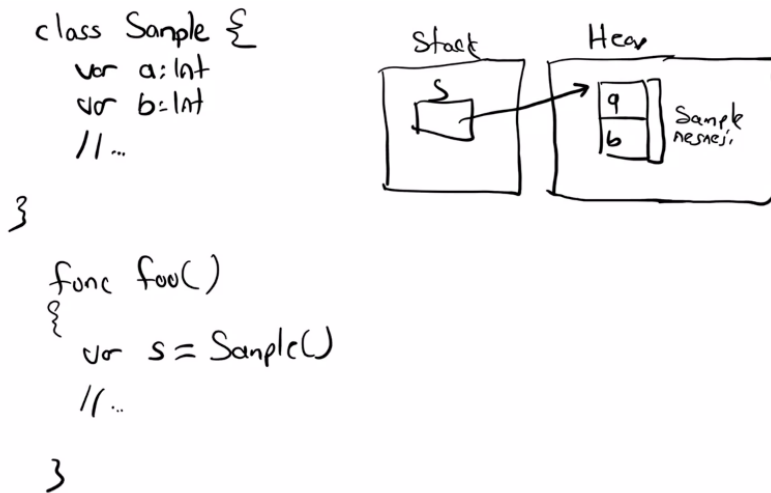
Swift'te fonksiyonların ve sınıfların dışında bildirilen değişkenler uygulamanın data bölümünde yaratılırlar. Data bölümü prosesin yaşamı boyunca (yani run time boyunca) bellekte kalmaktadır. Yani global değişkenler program yüklendiğinde bellekte yaratılırlar, program sonlanana kadar orada kalırlar.

Fonksiyonların ve metotların parametre değişkenleri ve onların blokları içerisinde bildirilmiş olan yerel değişkenler stack'te yaratılmaktadır. Stack doldur-boşalt tarzı bir alandır. Bir değişken programın akışı fonksiyon ya da metotta o değişkenin bildirildiği yere geldiğinde stack'te yaratılır. Fonksiyon ya da metot bittiğinde stack'ten yok edilir. Örneğin:



Görüldüğü gibi bir fonksiyon ya da metod çağrılmadığı sürece onların içerisindeki yerel değişkenler zaten bellekte yer kaplıyor durumda değildir. Stack'te değişkenlerin yaratılması ve yok edilmesi çok hızlı bir biçimde yapılmaktadır.

Sınıf nesneleri heap'te yaratılmaktadır. Sınıf nesneleri ile onların adreslerinin tutulduğu değişkenleri (referansları) birbirine karıştırmayınız. Nesnenin kendisi (yani onun veri elemanları) heap'te tutulmaktadır. Ancak onun adresini tutan referanslar stack'te olabilirler. Örneğin Sample bir sınıf olsun:



Bu örnekte foo içerisindeki s değişkeni stack'te hızlı bir biçimde yaratılmaktadır. Halbuki Sample() ifadesiyle Sample nesnesi heap'te yaratılır. Heap'teki yaratım stack'teki yaratıma göre görece olarak oldukça yavaştır. Nesnenin adresini tutan stack'teki s değişkeni fonksiyon sonlandığında otomatik olarak yok edilecektir. Fakat bu

değişkenin gösterdiği heap'teki nesne onu hiçbir referans göstermediği zaman sistem tarafından otomatik olarak silinmektedir. Belli bir anda bir sınıf nesnesini n tane referans gösteriyor durumdadır. Buna nesnenin referans sayacı denir. Nesnenin referans sayacı sıfıra düştüğünde artık nesne çöp durumuna gelir. Swift'teki ARC (ya da C# ve Java'daki garbage collector) nesneyi heap'ten siler.

Türetme Durumunda init Metotlarının Çağırılması

Bilindiği gibi C++' C# ve Java gibi dillerde türemiş sınıf türünden bir nesne yaratıldığında türemiş sınıfın başlangıç metotları (constructors) çağrılmaktadır. Bu dillerde türemiş sınıfın başlangıç metotları taban sınıfın başlangıç metotlarını çağırılmaktadır. Böylece nesnenin taban sınıf kısmına taban sınıfın başlangıç metotları ilkdeğerlerini atamaktadır. Halbuki Swift'te diğer nesne yönelimli dillerin çoğunda olduğu gibi türemiş sınıf başlangıç metodu (yani init metodu) otomatik olarak taban sınıfın başlangıç metodunu (yani init metodunu) çağırılmaz. Örneğin:

```
class A {
    var x: Int

    init(x: Int)
    {
        self.x = x
    }
}

class B : A {
    var y: Int = 20

    init(y: Int)          // error taban sınıfın x property'si değer almamış
    {
        self.y = y
    }
}

var b = B(y: 100);
```

Swift'te türemiş sınıfın taban sınıf kısmındaki property'lere ilkdeğerlerinin verilmesi için başvurulanan normal yol türemiş sınıfın taban sınıfın init metodunu super.init(...) ifadesi ile çağırmasıdır. Bu bakımdan Swift'te türemiş sınıflar için iki farklı init metodu tanımlanmıştır: "designated init metodu (designated initializer)" ve "convenience init metodu (convenience initializer)". Eğer türemiş sınıfın init metodunda convenience anahtar sözcüğü belirtilmemişse bu init metodu "designated" init metodu, convenience anahtar sözcüğü belirtilmişse bu init metodu da "convenience" init metodudur. Yani başka bir deyişle init metodunda hiçbir şey belirtilmediyse default olarak bu init metodu "designated init metodu" biçimindedir. Örneğin:

```
class A {
    //...
}

class B : A {

    init(a: Int)          // designated init metodu
    {
        //...
    }

    convenience init()    // convenience init metodu
    {
        //...
    }
}
```

```

{
    //...
}
//...
}

```

Swift'te sınıfın bir init metodu kendi sınıfının init metodunu self anahtar sözcüğüyle, taban sınıfın init metodunu ise super anahtar sözcüğüyle çağırmak zorundadır.

Türemiş sınıf ve taban sınıf init metotlarının yazımı sırasında şu kurallara uyulmak zorundadır:

1) Türemiş sınıfın "designated init" metodu taban sınıfın "designated init" metotlarından birini çağırmak zorundadır. Türemiş sınıfın "designated init" metotları kendi sınıflarının init metotlarını çağıramaz.

2) Türemiş sınıfın "convenience init" metodu aynı sınıfın başka bir init metodunu çağırmak zorundadır. Taban sınıfın herhangi bir init metodunu çağıramaz. Tabii türemiş sınıfın "convenience init" metodu sınıfın başka bir "convenience init" metodunu çağırıyor olabilir. Ancak bu zincirin en sonunda türemiş sınıfın "convenience init" metodunun türemiş sınıfın "designated init" metodunu çağırıyor olması gerekir.

Örneğin:

```

class A {
    var x: Int

    init(x: Int)
    {
        self.x = x
    }
}

class B : A {
    var y: Int

    init(x: Int, y: Int)
    {
        self.y = y

        super.init(x: x)
    }

    convenience init()
    {
        self.init(x: 10, y: 20)
    }
}

class C : B {
    var z: Int

    init(x: Int, y: Int, z: Int)
    {
        self.z = z

        super.init(x: x, y: y)
    }
}

```

```
}
```

```
let c = C(x:10, y: 20, z: 30)
```

3) Türemiş sınıfın "designated init metodu" ancak kendi sınıfının tüm veri elemanlarına değer atandıktan sonra taban sınıfın "designated init" metodunu çağırabilir. Yani taban sınıfın init metodu çağırılmadan önce türemiş sınıfın tüm veri elemanlarına değer atanmış olmak zorundadır. Örneğin:

```
class A {
    var x: Int

    init(x: Int)
    {
        self.x = x
    }
}

class B : A {
    var y: Int

    init(x: Int, y: Int)
    {
        super.init(x: x)    // error! henüz y'ye değer atanmamış
        self.y = y
    }
}
```

Burada B sınıfının y elemanına henüz değer atanmadan A sınıfının init metodu çağırılmıştır. Bu durum error oluşturur. Bu kural C++, Java ve C# gibi dillerdeki olağan akışla çelişkili gibi görünebilir. Gerçekten de o dillerde her zaman önce nesnenin taban sınıf kısmı ilkdeğerlenmektedir. Swift'te init metotları da override edildiği için böyle bir kurala gereksinim duyulmuştur.

4) Türemiş sınıfın "designated init" metodu taban sınıfın designated init metodunu çağırılmadan önce taban sınıfın veri elemanlarını ya da metotlarını kullanamaz. Kullanırsa error oluşur. Örneğin:

```
class A {
    var x: Int

    init(x: Int)
    {
        self.x = x
    }

    func foo()
    {
        //...
    }
}

class B : A {
    var y: Int

    init(x: Int, y: Int)
    {
        self.y = y
        super.x = 50    // error! taban sınıfın init metodundan önce property'si kullanılmış
    }
}
```

```

        super.init(x: x)
    }
}

```

5) Türemiş sınıfın "convenience init" metodu kendi sınıfının başka bir init metodunu çağırmadan önce kendi sınıfının ya da taban sınıfın bir veri elemanını ya da metodunu kullanamaz. Örneğin:

```

class A {
    var x: Int

    init(x: Int)
    {
        self.x = x
    }
}

class B : A {
    var y: Int

    init(x: Int, y: Int)
    {
        self.y = y

        super.init(x: x)
    }

    convenience init()
    {
        self.y = 50 // error!
        self.init(y: 10)
    }

    convenience init(y: Int)
    {
        self.init(x: 100, y: y)
    }
}

```

6) Türemiş sınıf için hiçbir init metodu yazılmamışsa taban sınıfın init metotları sanki türemiş sınıfın init metotlarıymış gibi kullanılır. Örneğin:

```

class A {
    var x: Int

    init(x: Int)
    {
        self.x = x
    }
}

class B : A {
    var y: Int = 20
}

var b: B = B(x: 10) // geçerli

```


7) Eğer türemiş sınıf taban sınıfın tüm designated init metotlarını override etmişse taban sınıfın tüm convenience init metotları sanki türemiş sınıfın convenience init metotlarıymış gibi kullanılabilir. Örneğin:

```
class A {
    var a: Int

    init(x: Int)
    {
        self.a = x
    }

    convenience init()
    {
        self.init(x: 10)
    }
}

class B : A {
    var b: Int

    override init(x: Int)
    {
        b = 20
        super.init(x: 10)
    }

    convenience init(x: Int, y: Int)
    {
        self.init()    // geçerli
        b = 20
    }
}

var b: B = B() // geçerli
```

Swift'te init metotları da override edilebilir. Bu durum override işleminin ele alındığı kısımda açıklanmaktadır. Yukarıdaki örnekte taban sınıfın convenience init metodu türemiş sınıfın convenience init metodu gibi kullanılmıştır. Benzer biçimde türemiş sınıfın convenience init metodu taban sınıfın convenience init metodunu doğrudan çağırmıştır.

Peki convenience ve designated init metotlarının ayrı ayrı olmasının anlamı nedir? Aslında şu durum sağlanmaya çalışılmıştır: Sınıfın bir init metodu diğer bir init metodunu çağırabilsin, fakat taban sınıfın init metodu toplamda yalnızca bir kez çağırılsın. İşte bunun derleme zamanında garanti altına alınması için bunlar uydurulmuştur. Aslında örneğin aynı sorun C#'ta benzer biçimde şöyle çözülmüştür. C#'ta başlangıç metotlarının kapanış parantezinden sonra ya `this(...)` ifadesi ya da `base(...)` ifadesi getirilir. Ancak iki ifade bir arada kullanılamaz. İşte aslında C#'ın bu anlamdaki `this(...)` sentaksı Swift'teki "convenience init" metoduna, `base(...)` sentaksı da "designated init" metoduna karşılık gelmektedir. Tabii burada belli bir "Objective-C" uyumu da aynı zamanda korunmaya çalışılmıştır.

Başarısız Olabilen init Metotları (Failable Initializers)

Bilindiği gibi nesne yönelimli pek çok dilde başlangıç metotlarının geri dönüş değerleri yoktur. Bu nedenle başlangıç metotlarında başarısızlıkla karşılaşıldığında genel olarak exception fırlatılmaktadır. Oysa Swift'te başarısız olabilen (failable) init metotları da vardır. Böyle init metotları init? ismiyle bildirilir.

```
class Sample {
    var msg: String = ""

    init?(msg: String)
    {
        if msg.isEmpty {
            return nil
        }
        self.msg = msg
    }
}

var s : Sample? = Sample(msg: "")

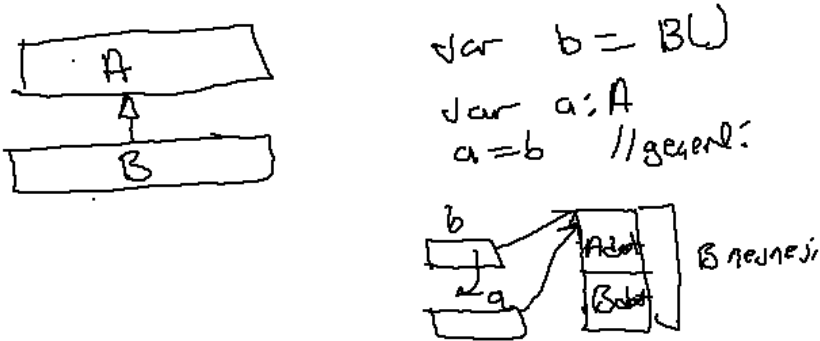
if let ss = s {
    print(ss)
}
else {
    print("nil")
}
```

Görüldüğü gibi başarısız olabilen init metotları ilgili sınıf T türündense aslında bize T? biçiminde seçenksel değer vermektedir. Örneğimizde init metoduyla yaratılan nesne referansının Sample? türüne atandığına dikkat ediniz.

Başarısız olabilen convenience ya da designated init metotları kendi sınıflarının ya da taban sınıflarının normal init metotlarını (non-failable init methods) çağırabilir. Ancak normal init metotları kendi sınıflarının ya da taban sınıflarının başarısız olabilen init metotlarını çağıramaz.

Türemiş Sınıftan Taban Sınıfa Referans Dönüştürmeleri

Farklı sınıflar türünden referanslar birbirlerine atanamazlar. Ancak bir istisna olarak nense yönelimli dillerin hepsinde türemiş sınıf türünden referans (ya da adres) taban sınıf türünden referansa (ya da göstericiye) doğrudan atanabilmektedir. Bunun nedeni türemiş sınıf nesnesinin taban sınıf elemanlarını da içeriyor olmasındandır. Örneğin:



Türemiş sınıf referansını taban sınıf referansına atadıktan sonra artık bu taban sınıf referansı bağımsız bir taban sınıf nesnesini gösteriyor durumda değildir. Türemiş sınıf nesnesinin taban sınıf kısmını gösteriyor durumdadır. Örneğin:

```
class A {
    var x: Int

    init(_ x: Int)
    {
        self.x = x
    }
}

class B : A {
    var y: Int

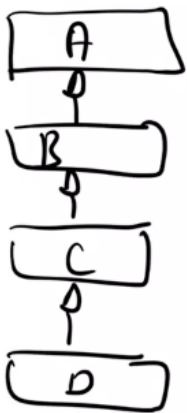
    init(_ x: Int, _ y: Int)
    {
        self.y = y
        super.init(x)
    }
}

var a: A
var b: B

b = B(10, 20)
a = b

print(a.x)    // 10
print(b.x)    // 10
print(b.y)    // 20
```

Bir türetme şeması içerisinde herhangi bir türemiş sınıf referansı herhangi bir taban sınıf referansına atanabilir. Örneğin:



Burada biz D sınıfı türünden bir referansı C sınıfı türünden, B sınıfı türünden ya da A sınıfı türünden bir referansa atayabiliriz. Örneğin:

```
class A {
    var x: Int
```

```

    init(_ x: Int)
    {
        self.x = x
    }
}

class B : A {
    var y: Int

    init(_ x: Int, _ y: Int)
    {
        self.y = y
        super.init(x)
    }
}

class C : B {
    var z: Int

    init(_ x: Int, _ y: Int, _ z: Int)
    {
        self.z = z
        super.init(x, y)
    }
}

class D : C {
    var k: Int

    init(_ x: Int, _ y: Int, _ z: Int, _ k: Int)
    {
        self.k = k
        super.init(x, y, z)
    }
}

var d = D(10, 20, 30, 40)
var a: A
var b: B
var c: C

c = d                // geçerli
print(c.x, c.y, c.z) // 10, 20, 30

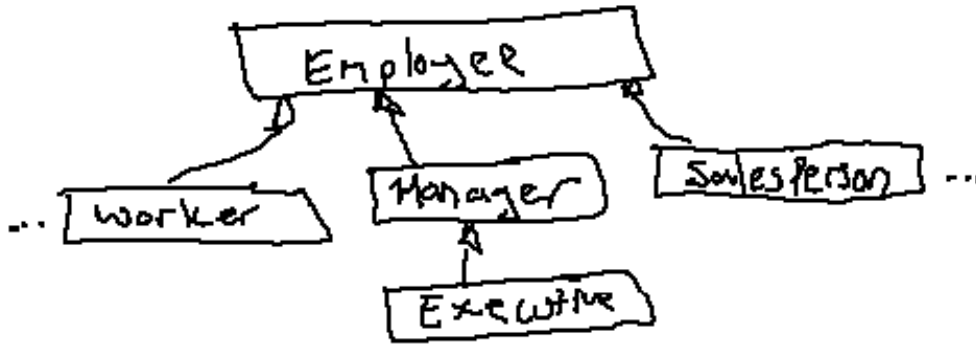
b = c
print(b.x, b.y)       // 10, 20

a = b
print(a.x)            // 10

a = d                // geçerli
print(a.x)            // 10

```

Türetilmiş sınıf referansının taban sınıf referansına atanabilmesinin önemli sonuçları vardır. Bu sayede bir türetme şeması üzerinde genel işlemler yapan fonksiyonlar ya da metotlar yazılabilir. Örneğin:



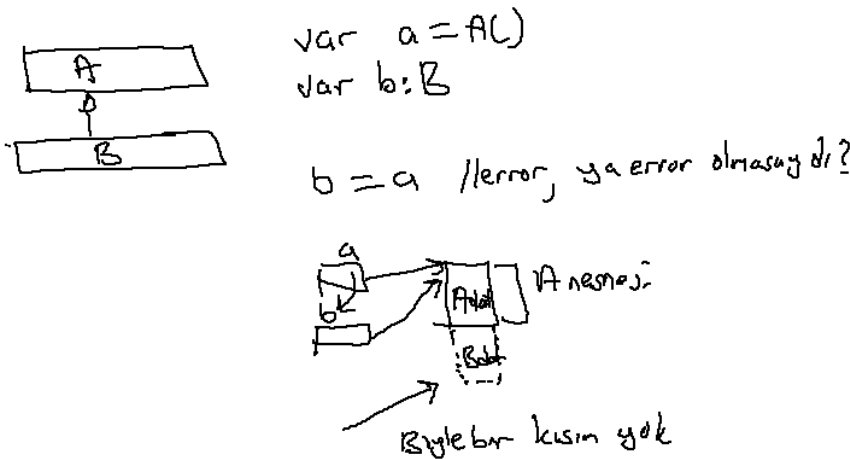
```

func displayInfo(e: Employee)
{
  //...
}
  
```

Biz burada displayInfo fonksiyonunu Employee türünden argümanla çağırmak zorunda değiliz. Worker, Manager, Executive, SalesPerson, ... türünden argümanlarla da çağırabiliriz. Böylece displayInfo genel işlem yapan bir fonksiyon durumuna gelmiştir. Tüm çalışan sınıflarının bir Employee kısmının olduğuna dikkat ediniz. Tabii burada biz displayInfo fonksiyonunu örneğin Executive argümanı ile çağırsak displayInfo Executive'in ancak Employee bilgilerini yazdırabilir. Taban sınıf referansı ile türemiş sınıfın elemanlarına erişmek (orada onlar olsa bile) mümkün değildir.

Türemiş sınıf referansının taban sınıf referansına atanabilmesi çokbiçimlilik (polymorphism) için gerekmektedir.

Yukarıdaki işlemin tersi yani taban sınıf referansının türemiş sınıf referansına atanması geçerli değildir. Eğer böyle bir atama yapılabilsen bu durumda olmayan veri elemanlarına erişme gibi bir durum oluşurdu. Örneğin:



Referansların Statik ve Dinamik Türleri

Referansların statik ve dinamik türleri vardır. (Ancak değer türlerinin bu biçimde iki türü yoktur.) Bir referansın statik türü bildirimde belirtilen türüdür. Dinamik türü ise referans bir nesneyi gösteriyorsa gösterdiği nesnenin bütünün türüdür. Örneğin:



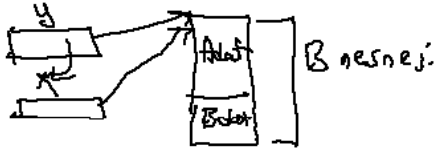
```
var x: A
var y: B = B()
```

```
x = y
```

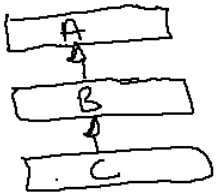
Burada x referansının statik türü A'dır, dinamik türü ise B'dir. Çünkü x'in gösterdiği yerde bütün B olan bir nesne vardır. x onun A parçasını göstermektedir.

```
var x: A
var y: B = B()
```

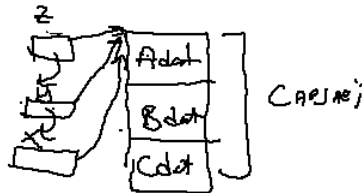
```
x = y
```



Burada y referansının statik türü de dinamik türü de B'dir. Örneğin:



```
var z = C()
var y: B
var x: A
y = z
x = y
```



Burada x'in statik türü A'dır. Fakat dinamik türü C'dir. Çünkü x'in gösterdiği yerdeki nesnenin bütünün türü (yani en geniş halinin türü) C'dir. Benzer biçimde y'nin statik türü B'dir fakat dinamik türü C'dir. z'nin ise hem statik hem de dinamik türü C'dir.

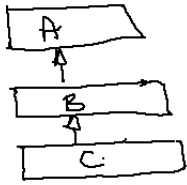
Referansın statik türü hiç değişmez. Fakat dinamik türü onun nereye gösterdiğine bağlı olarak değişebilir. Örneğin:

```

var x: A
x = A()
→ ①
x = B()
→ ②
x = C()
→ ③

```

Burada ok ile gösterilen üç yerde de x'in statik türü A'dır. Ancak dinamik türü sırasıyla A, B ve C olacak biçimde değişmektedir. Örneğin:



```

func foo(a: A)
{
    // a'nın dinamik türü?
}

var a = A = A()
var b = B = B()
var c = C = C()

foo(a) // foo'daki a'nın dinamik türü A
foo(b) // foo'daki a'nın dinamik türü B
foo(c) // foo'daki a'nın dinamik türü C

```

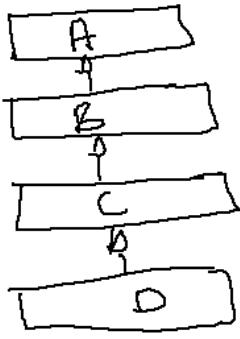
Burada foo'nun her çağırımında parametre değişkeni olan a'nın dinamik türü değişmektedir.

Taban Sınıftan Türemiş Sınıfa Referans Dönüştürmeleri (Downcating)

Bazen taban sınıftan türemiş sınıfa dönüştürme de yapılmak istenebilir. Örneğin bir fonksiyon ya da metot bazı gerekçelerden dolayı türemiş sınıf nesnesinin adresini bize taban sınıf referansıymış gibi veriyor olabilir. Bizim orijinal nesneyi tam olarak kullanabilmemiz için onu türemiş sınıfa dönüştürmemiz gerekir.

Taban sınıftan türemiş sınıfa yapılan dönüştürmeler haklı ya da haksız olabilir. Eğer dönüştürülmek istenen referansın dinamik türü dönüştürülmek istenen sınıfı kapsıyorsa dönüştürme haklıdır, kapsamıyorsa haksızdır.

Örneğin:



var a: A = C()
a'nın B'ye dönüştürülmesi haksızdır
a'nın D'ye dönüştürülmesi haksızdır

Aşağıdan yukarıya doğru dönüştürmelerin (upcasts) haksız olma olasılığı yoktur. O nedenle biz türemiş sınıf referansını doğrudan taban sınıf referansına atayabiliriz.

Swift'te aşağıya doğru dönüştürmeler otomatik olarak yapılmazlar. Yani biz taban türünden bir referansı türemiş sınıf türünden bir referansa doğrudan atayamayız. Swift'te aşağıya doğru dönüştürmeler as! ve as? operatörleriyle yapılmaktadır. Bu iki operatör de iki operandlı aralık operatörlerdir. Bu operatörlerle aşağıya doğru dönüştürme yapıldığında her zaman derleme aşamasından başarıyla geçilir. Ancak as! operatörü programın çalışma zamanı sırasında da kontrol uygulamaktadır. as! ve as? operatörlerinin sol tarafındaki operand bir sınıf türünden referans belirtmek zorundadır. Bunların sağ tarafındaki operand bir sınıf ismi olmak zorundadır. Örneğin:

```
class A {
    var x: Int

    init(_ x: Int)
    {
        self.x = x
    }
}

class B : A {
    var y: Int

    init(_ x: Int, _ y: Int)
    {
        self.y = y
        super.init(x)
    }
}

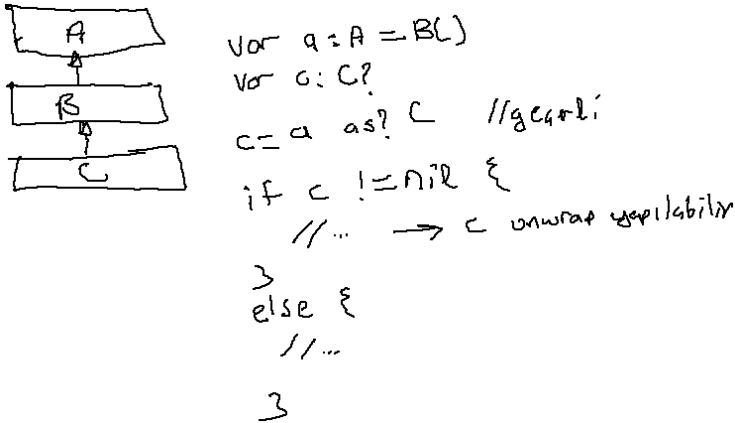
var x: A
var y: B = B(10, 2)
var z: B

// Burada x referansının dinamik türü B'dir

x = y                // geçerli (upcast)
z = x                // error: downcat as! ya da as? operatörü ile yapılmalı
z = x as! B          // geçerli
```


as! operatörü eğer dönüştürme haksızsa dönüştürmenin yapıldığı yerde exception oluşturur. Yani program çöker.

as? operatörü as! operatörüyle aynı biçimde kullanılır. Ancak bu operatör dönüştürme haksızsa exception fırlatmaz nil referans üretir. Anımsanacağı gibi Swift'te biz Java ve C#'ta olduğu gibi referanslara nil (onlarda nul) yerleştiremeyiz. Swift'te Referanslara dahi nil değer yerleştirilebilmesi için o türün seçeneysel (optional) olması gerekmektedir. Dolayısıyla as? operatörü de sağ tarafındaki tür (yani dönüştürülmek istenen tür) T ise bize T? türünden bir değer üretmektedir. Örneğin:



Örneğin:

```
class A {
    var x: Int

    init(_ x: Int)
    {
        self.x = x
    }
}

class B : A {
    var y: Int

    init(_ x: Int, _ y: Int)
    {
        self.y = y
        super.init(x)
    }
}

class C : B {
    var z: Int

    init(_ x: Int, _ y: Int, _ z: Int)
    {
        self.z = z
        super.init(x, y)
    }
}

var x: A = B(10, 20)

if let z = x as? C {
```

```

    print(z.x, z.y, z.z)
}
else {
    print("Dönüştürme haksız!")
}

if let y = x as? B {
    print(y.x, y.y)
}
else {
    print("Dönüştürme haksız!")
}

```

Swift'te de (tıpkı Java ve C#'ta olduğu gibi) aralarında türetme ilişkisi olmayan iki sınıf arasında as! ya da as? operatörleriyle dönüştürme yapılamaz. Örneğin:



Burada biz B'den C'ye dönüştürme yapamayız. Yapmaya çalışırsak derleme zamanında error oluşur. Çünkü aslında hiçbir zaman örneğin B türünden bir referans C'yi gösterir duruma gelmemektedir. Tabii eğer böyle bir şey B'den A'ya sonra A'dan C'ye dönüştürme yoluyla yapılamaya çalışılabilir. Ancak bu dönüştürme hiçbir zaman zaten haklı olamayacaktır.

is Operatörü

is operatörü iki operandlı araek bir operatördür. Bu operatörün sol tarafındaki operand bir referans, sağ tarafındaki operand bir sınıf ismi olmak zorundadır. Operatör sol taraftaki referansın dinamik türünün sağ taraftaki türü içerip içermediğine bakar. Eğer sol taraftaki referansın dinamik türü sağ taraftaki türü içeriyorsa true değeri, içermiyorsa false değeri üretir. Bu operatörün ürettiği değer Bool türündendir. Örneğin:

```

class A {
    var x: Int

    init(_ x: Int)
    {
        self.x = x
    }
}

class B : A {
    var y: Int

    init(_ x: Int, _ y: Int)
    {
        self.y = y
        super.init(x)
    }
}

```

```

var x: A = B(10, 20)

if x is B {
    let y = x as! B
    print("\(y.x), \(y.y)")
}
else {
    print("x'in dinamik türü B'yi içermiyor")
}

```

Taban ve Türemiş Sınıflarda Aynı İsimli Veri Elemanlar ve Override İşlemi

Swift'te taban ve türemiş sınıflarda aynı isimli veri elemanlarının bulunması uygun görülmemiştir. Benzer biçimde aynı isimli, aynı parametre türlerine, aynı etiketlere ve aynı geri dönüş değerine sahip metotlar taban ve türemiş sınıfta bir arada bulunamazlar. Ancak parametre türlerinden ya da etiketlerden en az biri farklı olan ya da geri dönüş değerleri farklı olan metotlar taban ve türemiş sınıflarda bir arada bulunabilirler. Örneğin:

```

class A {
    var x: Int = 10
    //...
}

class B : A {
    var x: Int = 20    // error!
    //...
}

```

Burada taban ve türemiş sınıflarda aynı isimli veri elemanları (stored properties) bulundurulmuştur. Bu durum error ile sonuçlanır. Örneğin:

```

class A {
    func foo(x y: Int)
    {
        //...
    }
    //...
}

class B : A {
    func foo(x y: Int)    // error!
    {
        //...
    }
    //...
}

```

Burada taban ve türemiş sınıfta aynı isimli aynı parametrik yapıya, aynı etikete ve aynı geri dönüş değeri türüne sahip metot bulundurulmuştur. Bu durum error ile sonuçlanır. Taban ve türemiş sınıflardaki metotların "overload" edilebilir biçimde olması gerekmektedir. Örneğin:

```

class A {
    func foo(a y: Int)
    {
        //...
    }
}

```

```

    }
    //...
}

class B : A {
    func foo(b y: Int)    // geçerli
    {
        //...
    }
    //...
}

```

Bu örnekte taban ve türemiş sınıflardaki foo metotları bir arada bulunabilir. Çünkü etiket farklılığı vardır. Örneğin:

```

class A {
    func foo(_ x: Int)
    {
        //...
    }
    //...
}

class B : A {
    func foo(_ x: Int, _ y: Int)    // geçerli
    {
        //...
    }
    //...
}

```

Burada da taban ve türemiş sınıftaki foo metotlarının parametrik yapıları farklıdır. Bunlar bir arada bulunabilirler.

Benzer biçimde taban ve türemiş sınıflarda aynı isimli static veri elemanları ve overload edilemeyen static metotlar da bulunamaz. Örneğin:

```

class A {
    static var x: Int = 10
}

class B : A {
    static var x: Int = 20    // error
    //...
}

```

Örneğin:

```

class A {
    static func foo(_ a: Int)
    {
        //...
    }
    //...
}

class B : A {

```

```

static func foo(_ a: Int) // error!
{
    //...
}
//...
}

```

Fakat örneğin:

```

class A {
    static func foo(_ a: Int)
    {
        //...
    }
    //...
}

class B : A {
    func foo(_ a: Int) // geçerli
    {
        //...
    }
    //...
}

```

Taban ve türemiş sınıfta aynı isimli elemanların bulunabilmesi için türemiş sınıfta o elemanların override anahtar sözcüğü ile belirtilmesi gerekir. Buna taban sınıftaki elemanın türemiş sınıfta override edilmesi denilmektedir.

Taban sınıftaki veri elemanları türemiş sınıfta ancak "computed property" olarak override edilebilir. Bu override işleminin yapılabilmesi için de taban sınıftaki veri elemanının var ile bildirilmiş olması gerekir. Eğer taban sınıftaki veri elemanı let ile bildirilmişse biz bunu türemiş sınıfta override edemeyiz. Taban sınıftaki var ile bildirilmiş veri elemanı override edilirken "computed property'de" hem get hem de set bölümü bulundurulmak zorundadır. Örneğin:

```

class A {
    var a: Int = 10
    //...
}

class B : A {
    var x: Int = 0
    override var a: Int { // geçerli
        get {
            return 20
        }
        set {
            x = newValue
        }
    }
    //...
}

```

Taban sınıftaki computed property de türemiş sınıfta computed property olarak override edilebilmektedir. Örneğin:

```

import Foundation

class A {
    var a: Double

    init(_ a: Double)
    {
        self.a = a
    }

    var val: Double {
        get {
            return a * a
        }
        set {
            self.a = sqrt(newValue)
        }
    }
}

class B : A {
    var b: Double

    init(_ a: Double, _ b: Double)
    {
        self.b = b
        super.init(a)
    }

    override var val: Double { // geçerli
        get {
            return b * b * b
        }
        set {
            self.b = pow(newValue, 1.0 / 3.0)
        }
    }
}

```

Burada taban sınıfın val isimli read/write computed property'si türemiş sınıfta override edilmiştir. Swift'te read/write (yani mutable) bir "computed property" read-only olarak override edilemez. Yani başka bir deyişle "computed property'nin" hem get hem de set bölümü varsa biz onu override ederken hem get hem de set bölümünü bulundurmamız zorundayız. Ancak read-only bir "computed property" read/write olarak override edilebilir. Başka bir deyişle yalnızca get bölümüne sahip bir property hem get hem de set bölümüne sahip olacak biçimde override edilebilmektedir. Örneğin:

Property gözlemcileri de override edilebilmektedir. Örneğin:

```

class A {
    var x: Double = 0 {
        willSet (newX) {
            print("A.x willSet(x): \(newX), \(x)")
        }
        didSet (oldX) {
            print("A.x didSet(x): \(oldX), \(x)")
        }
    }
}

```

```

}

class B : A {
    override var x: Double {
        willSet (newX) {
            print("B.x willSet(x): \ (newX), \ (x)")
        }
        didSet (oldX) {
            print("B.x didSet(x): \ (oldX), \ (x)")
        }
    }
}

```

Taban sınıftaki static veri elemanları ve metotlar türemiş sınıfta override edilememektedir.

Swift'te Çokbiçimlilik (Polymorphism)

Çokbiçimlilik biyolojiden aktarılmış bir terimdir. Biyolojide çokbiçimlilik canlıların çeşitli doku ve organlarının temel işlevi aynı kalmak üzere farklılık göstermesidir. Örneğin kulak pek çok canlıda vardır. Temel işlevi duymaktır. Fakat her canlı aynı biçimde duymaz.

Yazılımda çok biçimlilik üç biçimde tanımlanabilmektedir:

- 1) Biyolojik Tanım: Çokbiçimlilik taban sınıfın belli bir metodunun türemiş sınıflar tarafından onlara özgü bir biçimde yeniden tanımlanmasıdır.
- 2) Yazılım Mühendisliği Tanımı: Çokbiçimlilik türden bağımsız kod parçalarının yazılması için bir tekniktir.
- 3) Aşağı Seviyeli Tanım: Çokbiçimlilik önceden yazılmış kodların sonradan yazılacak kodları çağırabilmesi özelliğidir. (Normalde bunun tam tersi olması beklenir)

Swift'te çokbiçimlilik tıpkı Java'da olduğu gibi default bir durumdur. Yani sınıfın statik olmayan her metodu başına final anahtar sözcüğü getirilmediyse sanal metot gibi davranmaktadır.

Swift'te taban sınıftaki static olmayan bir metodun türemiş sınıfta aynı isimle, aynı geri dönüş değeri türüyle ve aynı parametrik yapıyla (yani aynı parametre türleri ve dışsal isimlerle) türemiş sınıfta bildirilmesine "taban sınıftaki metodun türemiş sınıfta override edilmesi" denilmektedir. Override işleminde ayrıca override anahtar sözcüğünün de türemiş sınıf metot bildiriminin başında bulundurulması gerekir. Örneğin:

```

import Foundation

class A {
    var a: Int

    init(a: Int)
    {
        self.a = a
    }

    func foo(val: Int) -> Int
    {
        print("A.foo")
    }
}

```

```

        return val + a
    }
}

class B : A {
    var b: Int

    init(a: Int, b: Int)
    {
        self.b = b
        super.init(a: a)
    }

    override func foo(val: Int) -> Int
    {
        print("B.foo")

        return val + b
    }
}

```

Burada taban sınıftaki foo türemiş sınıfta override edilmiştir. Türemiş sınıftaki override anahtar sözcüğü okunabilirliği artırmak amacıyla zorunlu tutulmuştur. (Java'da bunun gerekmediğine, C#'ta ise sanallığın virtual anahtar sözcüğü ile başlatıldığını anımsayınız. Eğer türemiş sınıfta override edilmek istenen metot taban sınıfta yoksa bu durum derleme aşamasında error oluşturur.

Tıpkı C++'ta olduğu gibi (Java ve C#'ta böyle değil) metot türemiş sınıfta başka bir erişim belirleyici anahtar sözcükle override edilebilir. (Örneğin taban sınıftaki public bir metot türemiş sınıfta private olarak override edilebilir.) Erişim belirleyici anahtar sözcükler ilerideki bölümlerde ele alınmaktadır.

Türemiş sınıfta override edilen bir metot ondan türetilecek sınıfta yeniden override edilebilir. Böylece override işlemi devam ettirilebilir. Örneğin:

```

class A {
    var a: Int

    init(a: Int)
    {
        self.a = a
    }

    func foo(val: Int) -> Int
    {
        print("A.foo")

        return val + a
    }
    //...
}

class B : A {
    var b: Int

    init(a: Int, b: Int)
    {
        self.b = b
    }
}

```



```

        super.init(a: a)
    }

    override func foo(val: Int) -> Int
    {
        print("A.foo")

        return val + b
    }
}

class C : B {
    var c: Int

    init(a: Int, b: Int, c: Int)
    {
        self.c = c
        super.init(a: a, b: b)
    }

    override func foo(val: Int) -> Int
    {
        print("C.foo")

        return val / c
    }
}

```

Taban sınıftaki metod türemiş sınıfta override edilmek zorunda değildir. Örneğin istenirse override işlemi daha alt sınıflarda da yapılabilir.



Burada taban A sınıfındaki foo metodu A'dan türetilmiş B'de override edilmemiştir fakat C'de override edilmiştir:

```

class A {
    var a: Int

    init(a: Int)
    {
        self.a = a
    }

    func foo(val: Int) -> Int
    {
        print("A.foo")

        return val + a
    }
}

```

```

    }
}

class B : A {
    var b: Int

    init(a: Int, b: Int)
    {
        self.b = b
        super.init(a: a)
    }
}

class C : B {
    var c: Int

    init(a: Int, b: Int, c: Int)
    {
        self.c = c
        super.init(a: a, b: b)
    }

    override fun foo(val: Int) -> Int
    {
        print("C.foo")

        return val / a
    }
}

```

Türemiş sınıfta override edilmiş bir eleman o sınıftan türetilen sınıflarda da override edilebilir. Örneğin:

```

class A {
    fun foo()
    {
        //...
    }
    //...
}

class B : A {
    override fun foo()    // geçerli
    {
        //...
    }
    //...
}

class C : B {
    override fun foo()    // geçerli
    {
        //...
    }
    //...
}

```

Tabii override işlemi içimn ilgili elemanın hemen doğrudan taban sınıfta olması gerekmez. Override edilecek eleman daha yukarılardaki bir sınıfta da olabilir. Örneğin taban A sınıfında bir metot ondan türetilmiş olan B sınıfında override edilmemiş olabilir ancak B'den türetilmiş C sınıfında override edilmiş olabilir:

```
class A {
    func foo()
    {
        //...
    }
    //...
}

class B : A {
    //...
}

class C : B {
    override func foo()    // geçerli
    {
        //...
    }
    //...
}
```

Sınıfın taban sınıflarında olmayan bir eleman override edilemez. Override zaten taban sınıfta olan elemanın türemiş sınıfta yeniden yazılması anlamına gelmektedir. Örneğin:

```
class A {
    //...
}

class B : A {
    override func foo() // error!
    {
        //...
    }
    //...
}
```

Swift'te init metotları da override edilebilir. Yani taban sınıfın init metodu türemiş sınıfta aynı parametrik yapıyla bildirilecekse türemiş sınıfta bunun da başına override anahtar sözcüğünün getirilmesi gerekir. Örneğin:

```
class A {
    init()
    {
        //...
    }
    //...
}

class B : A {
    init()    // error!
    {
        super.init()
        //...
    }
}
```

```
    //...
}
```

Burada türemiş sınıf init metodunda override anahtar sözcüğünün yazılması gerekirdi. Örneğin:

```
class A {
    init()
    {
        //...
    }
    //...
}

class B : A {
    override init()           // geçerli
    {
        super.init()
        //...
    }

    init(_ a: Int)           // geçerli
    {
        super.init()
        //...
    }
    //...
}
```

Belli bir elemanın türemiş sınıflarda override edilmesi engellenebilir. Bunun için final anahtar sözcüğü kullanılmaktadır. Örneğin:

```
class A {
    final func foo()
    {
        //...
    }
    //...
}

class B : A {
    override func foo()      // error!
    {
        //...
    }
    //...
}
```

Bu örnekte taban sınıftaki metod final anahtar sözcüğüyle bildirilmiştir. Artık bu metod türemiş sınıflarda override edilemez. Benzer biçimde veri elemanları da final olabilir. Örneğin:

```
class A {
    final var a: Int = 10
    //...
}

class B : A {
```

```

    override var a: Int {           // error!
        get {
            return 0
        }
        set {
            //...
        }
    }
    //...
}

```

Swift'te de tıpkı Java'da olduğu gibi (C#'ta bunun için sealed anahtar sözcüğü kullanılmaktadır) bir sınıf bildiriminin önüne de final anahtar sözcüğü getirilebilir. Bu durumda sınıftan türetme yapılamaz. Örneğin:

```

final class A {
    init()
    {
        //...
    }
    //...
}

class B : A {           // error!
    //...
}

```

Tabii gerek elemanlarda gerekse sınıflarda final belli bir türetmeden sonra da kullanılabilir. Örneğin:

```

class A {
    func foo()
    {
        //...
    }
    //...
}

class B : A {
    override func foo()      // geçerli
    {
        //...
    }
    //...
}

class C : B {
    final override func foo()    // geçerli
    {
        //...
    }
    //...
}

class D : C {
    override func foo()          // error!
    {
        //...
    }
}

```

```
    //...
}
```

Burada A taban sınıfındaki foo B override edilmiştir. Fakat C'de hem final hem de override edilmiştir. Artık C sınıfından türetilmiş olan D sınıfında bu foo metodu daha fazla override edilemez. Örneğin:

Benzer durum sınıflar için de geçerlidir. Örneğin:

```
class A {
    //...
}

final class B : A {    // geçerli
    //...
}

class C : B {          // error!
    //...
}
```

Çokbiçimli Mekanizma

Bir sınıf referansı ile bir metod çağrıldığında metod referansın static türüne ilişkin sınıfın faaliyet alanında aranır (yani önce o sınıfta sonra o sınıfın taban sınıflarında arama yapılır). Metod bulunursa bulunan metodun dinamik türüne ilişkin override edilmiş metod çağrılır. Örneğin:

```
class A {
    func foo()
    {
        print("A.foo")
    }
}

class B : A {
    override func foo()
    {
        print("B.foo")
    }
}

let a: A = B()

// a'nın statik türü A, dinamik türü B

a.foo()    // B.foo çağrılır
```

Örneğin:

```
class A {
    var a: Int

    init(a: Int)
    {
        self.a = a
    }
}
```

```

func foo(val: Int) -> Int
{
    print("A.foo")

    return val + a
}

class B : A {
    var b: Int

    init(a: Int, b: Int)
    {
        self.b = b
        super.init(a: a)
    }

    override func foo(val: Int) -> Int
    {
        print("B.foo")

        return val * b
    }
}

class C : B {
    var c: Int

    init(a: Int, b: Int, c: Int)
    {
        self.c = c
        super.init(a: a, b: b)
    }

    override func foo(val: Int) -> Int
    {
        print("C.foo")

        return val / c
    }
}

var a: A = C(a: 10, b: 20, c: 30)
var result: Int

result = a.foo(val: 90)    // C'nin
print(result)             // 3

```

Burada a referansının static türü A, dinamik türü C'dir. Dolayısıyla a.foo çağrısında dinamik türe ilişkin C sınıfının foo metodu çağırılmıştır. Örneğin:

```

import Foundation

class A {
    var a: Int

    init(a: Int)

```

```

    {
        self.a = a
    }

    func foo(val: Int) -> Int
    {
        print("A.foo")

        return val + a
    }
}

class B : A {
    var b: Int

    init(a: Int, b: Int)
    {
        self.b = b
        super.init(a: a)
    }

    override func foo(val: Int) -> Int
    {
        print("B.foo")

        return val * b
    }
}

class C : B {
    var c: Int

    init(a: Int, b: Int, c: Int)
    {
        self.c = c
        super.init(a: a, b: b)
    }

    override func foo(val: Int) -> Int
    {
        print("C.foo")

        return val / c
    }
}

func test(a: A, _ b: Int)
{
    let result = a.foo(b)
    print(result)
}

var a = A(a: 10)
var b = B(a: 10, b: 20)
var c = C(a: 10, b: 20, c: 30)

test(a, 100)
test(b, 100)
test(c, 100)

```


Burada test fonksiyonu her çağrıldığında parametre değişkeni olan a referansının dinamik türü farklı olmaktadır.

Eğer referansın dinamik türüne ilişkin sınıfta ilgili metot override edilmemişse yukarıya doğru o metodun override edildiği ilk taban sınıfın metodu çağrılır.

Swift'te init metotlarının önüne required anahtar sözcüğü getirilirse o sınıftan türetilen türemiş sınıflar o init metodunu override etmek zorundadır. Ayrıca bu durumda türemiş sınıfın override bildiriminde de required anahtar sözcüğünün bulundurulması gerekir. Örneğin:

```
class A {
    required init(a: Int)
    {
        //...
    }
    //...
}

class B : A {
    init()
    {
        super.init(a: 0)
        //...
    }

    required init(a: Int)
    {
        //..
        super.init(a: 0)
    }
    //...
}
```

required init metotları türemiş sınıfta yazılırken artık override anahtar sözcüğünün kullanılmasına gerek kalmamaktadır. Zaten required anahtar sözcüğü override anlamını da kendisi vermektedir. (Swift derleyicinde türemiş sınıfta hem required hem override belirleyicileri bulundurulursa "warning" oluşmaktadır.) required belirleyicisi yalnızca init metotlarında kullanılmaktadır. init dışındaki metotlar required yapılamazlar.

Yapılar türetmeye kapalı olduğu için yapıların init metotlarında required belirleyicisi kullanılamaz.

Override İşleminin Engellenmesi ve final Anahtar Sözcüğü

Daha önceden de belirtildiği gibi Swift'te metotlar tıpkı Java'da olduğu gibi default durumda override edilebilecek biçimdedir (yani sanal biçimdedir). Default sanallık performans konusunda küçük bir dezavantaj oluşturabilmektedir. Biz bir metodun türemiş sınıf tarafından override edilemeyeceğini final anahtar sözcüğü ile belirtebiliriz. Metodun override edilmeyeceği bilgisi derleyicinin daha etkin kod üretmesine yardımcı olabilmektedir. Anımsanacağı gibi bu işlem Java'da da aynı biçimde yapılmaktadır. Örneğin:

```
class A {
    //...
    final func foo()
    {
```

```

    //...
}
}

class B : A {    // B sınıfında artık foo override edilemez
    //...
}

```

Sınıfların init metotları final yapılamazlar. Benzer biçimde yapılar da türetmeye kapalı oldukları için onların metotları da final yapılamamaktadır.

Bir sınıfın tamamı final yapılabilir (Java'da da aynı biçimde sınıfın final yapılabilirdiğini anımsayınız.) Bu durumda o sınıftan türetme yapılamaz (yani C#'taki sealed sınıflar gibi). Türetmenin final ile engellenmesi o sınıf türünden nesneler yaratıldığında çokbiçimli mekanizmanın ortadan kalkmasından dolayı daha etkin kod üretilmesini sağlayabilmektedir. Aynı zamanda final sınıflar okunabilirliği de artırır.

Swift'te abstract Metotlar ve Sınıflar

Swift'te C++, Java ve C# (C++'ta ismi saf sanal metotlardır) olduğu gibi resmi bir abstract metot kavramı yoktur. Diğer dillerdeki abstract sınıflar Swift'te dolaylı olarak oluşturulabilir. Bu dolaylı oluşturma yöntemi eklenti metotlar (extension methods) ve protokoller konusunda ele alınacaktır.

Any ve AnyObject Türleri

Anımsanacağı gibi Java ve C#'ta tüm sınıflar (C#'ta aynı zamanda yapılar da) Object isimli bir taban sınıftan türetilmiş durumdadır. O dillerde biz bir sınıfı hiçbir sınıftan türetmesek bile onun Object sınıfından türetildiği varsayılmaktadır. Böylece o dillerde Object her türlü referansı atabileceğimiz bir tür işlevi görür. Ancak Swift - C++ olduğu gibi- tepede bir sınıftan türetilmiş sınıflara sahip bir dil değildir. Fakat Swift'te de her türden nesneyi atayabileceğimiz Any isimli bir tür bulunmaktadır. Any türünden nesne yaratılamaz ancak referanslar bildirilebilir ve her türden nesne any türünden referansa atanabilir. Örneğin:

```

class A {
    //...
}

struct B {
    //...
}

var x: Any

x = 123    // geçerli
x = 12.3   // geçerli
x = A()    // geçerli
x = "ali"  // geçerli
x = B()    // geçerli

```

Any türünden değişkene yalnızca referans türlerini değil yapı değişkenlerini de atayabildiğimize dikkat ediniz. (Swift'te Int, String gibi türlerin yapı belirttiğini anımsayınız. Any türüne bir yapı ya da enum değeri atandığında kutulama dönüştürmesi yoluyla bu nesnenin heap'te bir kopyası çıkartılmakta ve any referansı artık o kopyayı göstermektedir.)

Örneğin bir fonksiyonun parametresi Any türünden olabilir. Bu durumda biz fonksiyonu herhangi bir türden değişkenle çağırabiliriz:

```
class A {
    //...
}

struct B {
    //...
}

func foo(_ a: Any)
{
    print(type(of: a))
}

foo(10)
foo(12.3)
foo("ali")
foo(A())
foo(B())
```

Swift'te Any türü Java ve C#'taki gibi bir taban sınıf değildir. Her türden değişkenin atanabildiği genel bir türdür.

Any türünden bir değişken herhangi bir işleme sokulamaz. Ancak as! ya da as? operatörleriyle orijinal türe dönüştürülüp işleme sokulmalıdır. Bu bakımdan Any türü Java ve C#'taki Object türüyle işlevsel olarak eşdeğer değildir.

```
class A {
    func foo()
    {
        print("A.foo")
    }
}

struct B {
    //...
}

var x: Any
var a: A
var i: Int
var b: B?
var d: Double?

x = A()
a = x as! A    // Dönüştürme haklı
a.foo()

x = 100
i = x as! Int   // Dönüştürme haklı
print(i)        // 100

x = A()
b = x as? B
```

```

if b != nil {
    print("Dönüştürme haklı")
}
else {
    print("Dönüştürme haksız")
}

x = 100
d = x as? Double
if d != nil {
    print("Dönüştürme haklı")
}
else {
    print("Dönüştürme haksız")
}

```

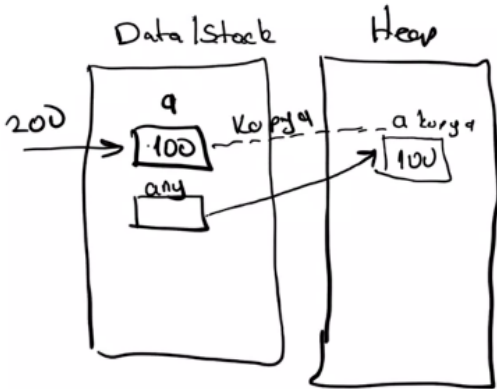
Peki any bir referans mıdır? Yanıt evet any bir referanstır. Swift'te bir referans stack'teki bir yapı nesnesini gösteremez. Ancak heap'teki bir nesneyi gösterebilir. O halde biz any türünden bir referansa bir yapı nesnesini atadığımızda any neyi tutmaktadır? İşte anımsanacağı gibi bu duruma C# dünyasında "kutulama dönüştürmesi (boxing conversion)" denilmektedir. Swift'te de bir yapıyı Any türünden bir referansa atadığımızda C#'taki kutulama dönüştürmesi gibi bir olay gerçekleşir. Yani o yapı değişkeninin heap'te bir kopyası oluşturulur. Any referansı o kopyayı gösterir hale gelir. Any türünden referansa atama yaptıktan sonra biz asıl nesneyi any referansının gösterdiği yerdeki kopya değiştirecektir. Örneğin:

```

var a = 100
var any: Any
any = a // Burada any a nesnesini göstermiyor, a'nın kopyasını gösteriyor
a = 200
print(any as! Int) // 100

```

Bu durumu şekilsel olarak şöyle gösterebiliriz:



Any türünden referanslara fonksiyon türlerini de atayabiliriz. Örneğin:

```

func foo()
{
    print("foo")
}

var any: Any

```

Örneğin:

```
any = foo
let f = any as! ()->>()
f()
```

Any türünden diziler de bildirilebilir. Bu durumda bir Any dizisinin elemanları farklı türlerden (heterojen) nesnelerin adreslerini tutuyor durumda olur. Biz de bu diziye dolaşarak dizinin elemanlarının türlerini is operatörü ile belirleyip uygun işlemler yapabiliriz. Örneğin:

```
let anys: [Any] = [10, 20, 30, "ali", "veli", "selami"]

for any in anys {
    if any is Int {
        let val = any as! Int
        print(val)
    }
    else if any is String {
        let s = any as! String
        print(s)
    }
}
```

Any türünden bir referansa seçeneysel bir türü de atayabiliriz. Bu duruma Swift derleyicileri uyarı vermektedir. Any türünden referansa seçeneysel bir tür atandığında yeniden as? ya da as! operatörü ile biz onu seçeneysel türe dönüştürmeliyiz. Örneğin:

```
var a: Int? = 10
var b: Any?
var c: Int?

c = b as! Int?
```

Ancak istenirse Any? türünden de referanslar bildirilebilir. Bu durumda bu referanslara biz nil de atayabiliriz.

AnyObject türü Any türüne benzerdir. Ancak AnyObject türüne yalnızca kategori olarak referans türlerine ilişkin nesneler atanabilir. Halbuki Any türüne her nesne atanabilmektedir. Örneğin:

```
class Sample {
    //...
}

var a: AnyObject

a = 100           // error!
a = "Ali"         // error!
a = [1, 2, 3]     // error!
a = Sample()      // geçerli
```

AnyObject türünün kullanımıyla özellikle Cocoa ortamında sıkça karşılaşacağız.

Otomatik Referans Sayacı (Automatic Reference Counting) Mekanizması ve Çöp Nesnelerin Yok Edilmesi

Swift'te Java ve .NET ortamlarındaki gibi bir çöp toplama mekanizması yoktur. Apple Objective-C'ye böyle bir mekanizmayı isteğe bağlı olarak eklemişse de bundan pişman olmuştur. Otomatik referans sayacı mekanizması işlev olarak çöp toplama mekanizmasına benzerdir. Yani her iki mekanizmanın da amacı heap'te artık kimsenin kullanmadığı çöp durumuna gelmiş olan nesnelerin otomatik yok edilmesidir. Ancak çöp toplama mekanizması ile otomatik referans sayacı mekanizmasının işletilmesinde önemli teknik farklılıklar vardır. Hangi mekanizmanın daha etkin olduğu konusunda da tartışmalar devam etmektedir. Ancak Apple'ın tercihi Swift'te "Otomatik Referans Sayacı (ORS)" olmuştur.

Belli bir anda heap'te yaratılmış olan bir nesneyi belli sayıda referans göstermektedir. Buna o nesnenin referans sayacı denir. Nesnenin referans sayacı nesnenin içerisinde tutulmaktadır. Her atama işleminde nesnenin referans sayacı bir artırılır. Yani nesnenin referans sayacı belli bir anda o nesnenin kaç referans tarafından gösterildiğini belirtmektedir. Örneğin:

```
var s = Sample()
var t = s
--->
```

Ok ile gösterilen noktada Sample nesnesi iki referans tarafından gösterilmektedir. O halde söz konusu Sample nesnesinin o noktadaki referans sayacı 2'dir. Nesnenin referans sayacı artabileceği gibi eksilebilir de. Nesnenin referans sayacı sıfıra geldiğinde artık nesneyi o hiçbir referans göstermiyor durumdadır. Zaten bu noktadan sonra nesneyi bir referansın göstermesi mümkün değildir. İşte Swift'te nesnenin referans sayacı sıfıra düşer düşmez ORS mekanizması yoluyla nesne heap'ten silinmektedir. Örneğin:

```
class Sample {
    var a: Int

    init()
    {
        a = 0
    }

    init(a: Int)
    {
        self.a = a
    }
    //...
}

func bar(_ s: Sample)
{
    // RS = 3
    let m = s
    // RS = 4
}

func foo()
{
    let s = Sample() // RS'si izlenecek nesne

    // RS = 1

    let k = s
}
```

```

    // RS = 2

    bar(k)

    // RS = 2
}

// RS = 0

foo()

// RS = 0

```

Bu örnekte foo çağrıldığında onun içerisinde yaratılan Sample nesnenin referans sayacı duruma göre artıp azalmıştır. foo metodunun çağırılması bitince o nesnenin referans sayacı sıfıra düşmüştür. İşte ORS mekanizması o nesnenin referans sayacı sıfıra düşer düşmez deterministik bir biçimde (yani zamanı ve yeri belirlenecek biçimde) nesneyi heap'ten yok eder.

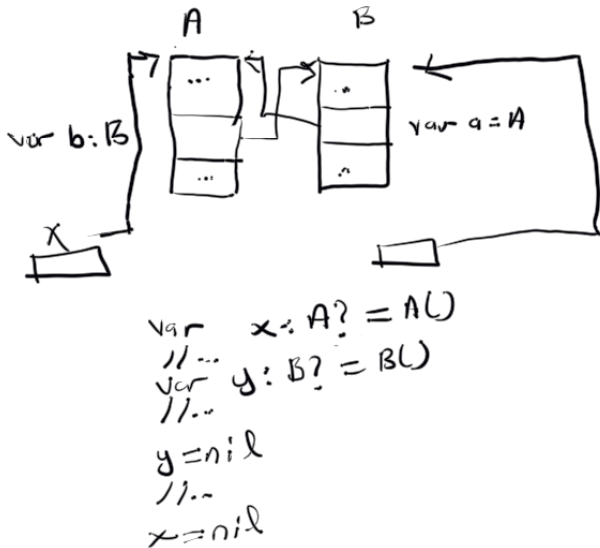
Bir nesnenin referans sayacının düşürülmesinin bir yolu da onu gösteren referansa nil değerini yerleştirmektir. Tabii bu durumda türün de seçeneysel olması gerekir. Örneğin:

```

var s: Sample? = Sample()
// nesnenin referans sayacı 1'dir.
s = nil
// artık nesnenin referans sayacı 0 durumdadır

```

ORS mekanizmasındaki önemli handikaplardan biri birbirini gösteren sınıf nesneleridir. Bu durumda bir döngüsellik oluşur. İki nesnenin de referans sayacı bir türlü sıfır olamaz.



Şekilden de görüldüğü gibi iki sınıf nesnesinin property'leri birbirlerini gösteriyor durumdaysa onları kullanan programcı kullandığı referansları nil yaparak ya da onların yok olmasını sağlayarak bu iki nesnenin yok edilmesine neden olamaz. İşte bu sorunu çözmek için dille zayıf (weak) referans ve sahihsiz (unowned) referans kavramları sokulmuştur.

weak bir referans seçeneksel türlerle bildirilebilir. Bunun için referans bildiriminin başına weak anahtar sözcüğü getirilir. Örneğin:

```
var s: Sample? = Sample()
weak var k: Sample?

k = s
```

Swift'te weak ya da unowned olmayan referanslara güçlü referanslar "strong references" de denilmektedir. Zayıf referanslar nesnenin referans sayacını artırmamaktadır. Yani yukarıdaki örnekte `k = s` atamasına rağmen nesnenin referans sayacı hala 1'dir. Dolayısıyla biz `k`'ya nil atayarak nesnenin referans sayacını da azaltamayız. Ancak `s`'ye nil atarsak nesnenin referans sayacını azaltıp sıfır yaparız:

```
var s: Sample? = Sample()
weak var k: Sample?

k = s
s = nil // nesne artık silinecek
```

Kuvvetli bir referansın gösterdiği yerdeki nesne yok edildiğinde sistem onu gösteren bütün zayıf referansların içerisine nil değerini yerleştirir. Örneğin:

```
var s: Sample? = Sample()
weak var k: Sample?

k = s
s = nil // nesne artık silinecek

print(k == nil ? "k == nil" : "k != nil")
```

Burada ekrana "`k == nil`" yazısı çıkacaktır.

Şüphesiz yeni yaratılan bir nesnenin adresinin zayıf bir referansa atanması anlamsızdır. Çünkü o nesne yaratılır yaratılmaz referans sayacı da artırılmadığı için yok edilecektir. Örneğin:

```
weak var s: Sample? = Sample() // anlamsız ama geçerli
```

Aşağıdaki kod parçasında counry referansına nil atadığımızda Country ve City nesneleri yok edilemeyecektir:

```
class City {
    var cityName: String
    var country: Country?

    init(cityName: String)
    {
        self.cityName = cityName
    }

    deinit {
        print("City deint")
    }
}
```



```

class Country {
    var countryName: String
    var capital: City

    init(countryName: String, capitalName: String)
    {
        self.countryName = countryName
        capital = City(cityName: capitalName)
        capital.country = self
    }

    deinit {
        print("Country deint")
    }
}

var country: Country? = Country(countryName: "Türkiye", capitalName: "Ankara")

country = nil           // Burada döngüsel durum yüzünden nesneler yok edilemeyecektir

```

country referansına nil atadığımızda hala Country nesnesini gösteren bir referans olduğu için Country nesnesinin referans sayacı 0'a düşmez, 1'de kalır. Böylece City nesnesinin referans sayacı da 0'a düşmemiş olur. İşte çözüm zayıf referans kullanmaktır. City sınıfındaki country referansı zayıf olarak bildirilirse bu döngüsel durum sorunu çözülür. Çünkü zayıf referans nesnenin referans sayacını artırmayacaktır. Bu durumda country referansı nil'e çekildiğinde artık Country nesnesini gösteren bir referans kalmamış olacaktır. Bu durumda da ORS Country nesnesini yok edecektir. Böylece City nesnesini de gösteren referans kalmadığından o da yok edilecektir.

Yukarıdaki örnekte her iki sınıftaki referansın da zayıf olamayacağına dikkat ediniz. Ayrıca kodun yazımına göre Country sınıfındaki capital referansı da zayıf olarak bildirilemez.

Sahipsiz (unowned) referanslar da zayıf referanslar gibi nesnenin referans sayacını artırmazlar. Ancak sahipsiz referanslar seçeneksel olmak zorunda değildir. Sahipsiz bir referansın gösterdiği yerdeki nesne yok edilirse sahipsiz referansın içindeki değer değişmez. Ancak bu adresin geçerliliği de yoktur. Dolayısıyla sahipsiz referansın gösterdiği yerdeki nesne yol edilmişse bizim artık o sahipsiz referansı kullanmamamız gerekir. Kullanırsak exception oluşur. Örneğin:

```

var s: Sample? = Sample()
unowned var k: Sample

k = s!
s = nil

k.a = 10           // exception oluşur

```

O halde zayıf (weak) referanslarla sahipsiz (unowned) referanslar arasındaki benzerlikler ve farklılıklar şunlardır:

- 1) Hem zayıf hem de sahipsiz referanslar nesnenin referans sayacını artırmazlar.
- 2) Zayıf referanslar seçeneksel türden olmak zorundadır. Ancak sahipsiz referanslar seçeneksel türden olamazlar.

3) Zayıf referansların gösterdiği nesne yok edildiğinde sistem zayıf referanslara nil değeri yerleştirir. Ancak aynı durum sahipsiz referanslar için geçerli değildir.

Sahipsiz referansları da aynı biçimde döngüsellikte kullanabiliriz.

Peki ne zaman zayıf referans ne zaman sahipsiz referans kullanmalıyız? Eğer referansımız seçeneksel olacaksa bu durumda mecburen zayıf referans, seçeneksel olmayacaksa sahipsiz referans kullanırız.

Sınıfların deinit Metotları

Nasıl Java ve C#'ın Finalize metotları, C++'ın destructor metotları varsa Swift'te de sınıfların deinit metotları vardır. Bir sınıf nesnesi ORS mekanizması tarafından heap'ten yok edilmeden hemen önce o nesne için o sınıfın deinit metodu çağrılır. deinit metodunun parametresi ve geri dönüş değeri yoktur. Bildiriminin genel biçimi şöyledir:

```
deinit {  
    //...  
}
```

Örneğin:

```
class Sample {  
    var a: Int  
  
    init()  
    {  
        a = 0  
  
        print("Sample init")  
    }  
  
    init(a: Int)  
    {  
        self.a = a  
  
        print("Sample init")  
    }  
  
    deinit {  
        print("Sample deinit")  
    }  
}  
  
var s: Sample? = Sample()  
print("test-1")  
s = nil  
print("test-2")
```

Swift'te tıpkı C++'ta olduğu gibi deinit metodu deterministiktir. Buradaki deterministik terimi deinit metodunun tam olarak hangi çağrılacağına belirli olması anlamına gelir. Halbuki Java ve C#'taki Finalize metotları deterministik değildir. Çünkü bu dillerde nesnenin referans sayacı 0'a düştüğünde nesnenin hangi süre sonra heap'ten yok edileceği belli değildir. Dolayısıyla Java ve C#'taki Finalize metotlarının tam olarak hangi noktada çağrılacağı belirsizdir. Maalesef o dillerde bu Finalize metotları nda onların bu özelliklerinden dolayı pek

çok şey yapılamamaktadır. Halbuki Swift'te nesnenin referans sayacı 0 olur olmaz hemen deinit metodu çağrılmaktadır.

İçerme ilişkisinde Swift'te içeren nesnenin referans sayacı 0 olunca hemen içeren nesne için deinit metodu çağrılır. Bu çağrıdan sonra içerilen nesne silinecek ve onun için de deinit metodu çağrılacaktır. Böylece biz Swift'te içeren nesnenin deinit metodu içerisinde içerilen nesne referansını kullanabiliriz. (Bunu Java ve C#'ta yapamayacağımızı anımsayınız).

Türemiş sınıfın init metodunda taban sınıfın init metodunun çağrılmasının programcının sorumluluğunda olduğunu anımsayınız. Fakat türetme durumunda türemiş sınıfın deinit metodu taban sınıfın deinit metodunu blok sonunda derleyicinin yerleştirmiş olduğu gizli bir çağırma kodu yoluyla otomatik olarak çağırılmaktadır. Bunun için programcının bir şey yapmasına gerek yoktur. Bu çağırılma biçimi tamamen C++'ta olduğu gibidir. Örneğin:

```
class A {  
    init()  
    {  
        print("A init")  
    }  
  
    deinit {  
        print("A deinit")  
    }  
}  
  
class B : A {  
    override init()  
    {  
        print("B init")  
  
        super.init()  
    }  
  
    deinit {  
        print("B deinit")  
    }  
}  
  
var b: B? = B()  
b = nil  
print("ok")
```

deinit metotları programcı tarafından çağrılmaz. Onları ya ORS mekanizması ya da yukarıda belirtildiği gibi türemiş sınıfın deinit metotları ana (bloğun sonunda) gizlice çağırılmaktadır.

Tabii Swift'te bir sınıfın deinit metodunun bulunması zorunlu değildir. Pekiyi türetme durumunda türemiş sınıfın deinit metodun programcı tarafından yazılmamışsa yine de taban sınıfın deinit metodu çağrılacak mıdır? Evet, bu tür durumlarda türemiş sınıf nesnesinin referans sayacı 0'a düştüğünde ORS mekanizması türemiş sınıfta bir deinit olmadığı için türemiş sınıfın deinit metodunu çağırmaya çalışmayacaktır. Ancak taban sınıfın deinit metodu yine çağrılacaktır.

Peki deinit metoduna neden gereksinim duyulmaktadır? İşte init metotlarında birtakım işlemler yapılmış olabilir ve bunların geri alınması gerekebilir. Örneğin init metodunda bir dosya açılmışsa bunun deinit metodunda kapatılması uygun olur. Bunun gibi init metodunda yaratılmış olan kaynakların nesne yok edildiğinde otomatik olarak yok edilmesi deinit metodunda yapılabilir.

Eklentiler (Extensions)

Bilindiği gibi nesne yönelimli programlama tekniğinde bir sınıfa onu değiştirmeden eleman ekleme işlemi türetme yoluyla yapılmaktadır. Ancak türetme işleminin belli bir kod maliyeti vardır. Ayrıca türetme işlemiyle yeni bir sınıf da oluşturulmaktadır. Programa yeni bir sınıfın eklenmesi bazı durumlar için algısal karmaşıklık artırıcı bir yük oluşturabilmektedir. (Ayrıca Swift'te yapıların ve enum türlerinin türetmeye kapalı olduğunu da anımsayınız) İşte eklentiler maliyetsiz olarak mevcut bir sınıf, yapı ya da enum türüne eleman eklemekte kullanılır. Fakat eklentilerin de türetmeye göre bazı kısıtları vardır.

Eklenti bildiriminin genel biçimi şöyledir:

```
extension <isim> {  
    //...  
}
```

Burada isim eklenti yapılacak sınıf, yapı ya da enum'un ismidir.

Eklentiler statik olan ya da olmayan metotlara ve hesaplanmış property'lere sahip olabilirler ancak depolanmış property'lere (yani veri elemanlarına) sahip olamazlar. Eklentiler içerisinde asıl sınıfın, yapının ya da enum'un tüm elemanları doğrudan kullanılabilir. Örneğin:

```
extension Double {  
    var mm: Double { return self / 1000}  
    var cm: Double { return self / 100 }  
    var m: Double { return self }  
    var km: Double { return self * 1000 }  
}  
  
var distance = 3.4.km  
print(distance)  
  
var result = 3.km + 400.m  
print(result)
```

Anahtar Notlar: Yukarıdaki örnekte 3.km ifadesindeki 3 Double derleyici tarafından Double olarak ele alınmaktadır. Halbuki resmi dokümanlarında bunu destekleyecek resmi bir kural göze çarpmamaktadır. Fakat yapılan denemelerde hem Double hem de Int yapıları için aynı eklenti bulunduğunda 3.km ifadesinde Int eklentisindeki km property'sinin seçildiği görülmektedir.

Örneğin:

```
struct Complex {  
    var real: Double  
    var imag: Double  
  
    init()  
    {  
        real = 0
```

```

        imag = 0
    }

    init(_ real: Double, _ imag: Double)
    {
        self.real = real
        self.imag = imag
    }
}

extension Complex {
    static func Add(_ z1: Complex, _ z2: Complex) -> Complex
    {
        var result = Complex()
        result.real = z1.real + z2.real
        result.imag = z1.imag + z2.imag

        return result
    }

    var description: String {
        return real.description + " + " + imag.description + "i"
    }
}

var z1 = Complex(3, 2)
var z2 = Complex(5, 3)
var result:Complex

result = Complex.Add(z1, z2)
print(result.description)

```

Eklentiler init metotları içerebilir. Ancak eklenti bir sınıf için oluşturulmuşsa eklentideki init metodu "designated" olamaz, "convenience" olmak zorundadır. Anımsanacağı gibi "designated" ve "convenience" init metotları sınıflar için söz konusudur ve designated init metodu taban sınıfın, convenience init metodu ise kendi sınıfının init metodunu çağırmak zorundadır. Eğer eklentiler yapılar için oluşturulmuşsa normal init metotlarına sahip olabilirler. Örneğin:

```

struct Complex {
    var real: Double
    var imag: Double

    init()
    {
        real = 0
        imag = 0
    }

    init(_ real: Double, _ imag: Double)
    {
        self.real = real
        self.imag = imag
    }
}

extension Complex {

```

```

init(_ real: Double)
{
    self.real = real
    self.imag = 0
}

static func Add(_ z1: Complex, _ z2: Complex) -> Complex
{
    var result = Complex()
    result.real = z1.real + z2.real
    result.imag = z1.imag + z2.imag

    return result
}

var description: String {
    var result = real.description

    if imag != 0 {
        result += " + " + imag.description + "i"
    }
    return result
}
}

var z = Complex(10)
print(z.description)

```

Eğer yukarıdaki Complex türü bir sınıf olarak düzenlenseydi eklentideki init metodu "convenience" olmak zorunda olurdu:

```

class Complex {
    var real: Double
    var imag: Double

    init()
    {
        real = 0
        imag = 0
    }

    init(_ real: Double, _ imag: Double)
    {
        self.real = real
        self.imag = imag
    }
}

extension Complex {

    convenience init(_ real: Double)
    {
        self.init(real, 0)
    }

    static func Add(_ z1: Complex, _ z2: Complex) -> Complex
    {
        let result = Complex()

```

```

        result.real = z1.real + z2.real
        result.imag = z1.imag + z2.imag

        return result
    }

    var description: String {
        var result = real.description

        if imag != 0 {
            result += " + " + imag.description + "i"
        }
        return result
    }
}

var z = Complex(10)
print(z.description)

```

Örneğin:

```

extension Int {
    func repeatition(_ f: (Int)->Void)
    {
        for i in 0..

```

Eklentiler subscript elemanlara da sahip olabilir. Örneğin:

```

extension Int {
    subscript(_ index: Int) -> Int {
        var m = 1
        for _ in 0..

```

Protokoller

Protokol terimi Objective-C'den aktarılmıştır. Protokoller Java ve C#'taki arayüzlere (interfaces) karşılık gelmektedir. Bilindiği gibi arayüzler çoklu türetmenin bazı avantajlarını dile sokmak düşünülmüş türlerdir. C++'ta zaten çoklu türetme olduğu için orada arayüzlere ayrıca gereksinim duyulmamıştır.

Protokol bildiriminin genel biçimi şöyledir:

```
protocol <isim> {
    //...
}
```

Bir protokol depolanmış property'lere (yani veri elemanlarına) sahip olamaz. Protokoller metotlara ve hesaplanmış property'lere sahip olabilir. Ancak protokol elemanları gövde içeremez. Örneğin:

```
protocol Test {
    func foo(_ a: Int) -> Int
    func bar()
}
```

Sınıflar, yapılar ve enum'lar protokolleri destekleyebilir. Protokol desteği için bildirimde sınıf, yapı ya da enum isminden sonra ':' atomunu protokol listesi izlemelidir. Görüldüğü gibi protokollerin desteklenme sentaksı türetme sentaksına benzemektedir. Örneğin:

```
class Sample : Test {
    //...
}
```

Anahtar Notlar: Swift terminolojisinde protokolü desteklemek İngilizce "conform" sözcüğü ile ifade edilmektedir. Anımsanacağı gibi aynı kavram Java ve C#'ta "implement" sözcüğüyle ifade ediliyordu. "Conform" sözcüğü İngilizce "uymak (özellikle kurallara vs.)" anlamına gelmektedir. Ancak biz kursumuzda falanca sınıfın ya da yapının "protokole uyması" yerine falanca sınıfın ya da yapının ilgili protokolü desteklemesi biçiminde ifade kullanacağız.

Bir sınıf, yapı ya da enum bir protokü destekliyorsa o protokülün bütün elemanlarını bildirmelidir. Yani örneğin protokolde bir metot varsa o protokolü destekleyen sınıf, yapı ya da enum o metodu gövdesiyle bildirmek zorundadır. Bu bildirim sırasında metodun geri dönüş değeri, etiket isimleri ve parametre türleri aynı olmak zorundadır. Ancak parametre değişkenlerinin isimleri aynı olmak zorunda değildir. Örneğin:

```
protocol Test {
    func foo(_ a: Int) -> Int
    func bar()
}

class Sample : Test {
    func foo(_ a: Int) -> Int
    {
        return a * a
    }

    func bar()
    {
        print("bar")
    }
}
```

Bir sınıf, yapı ya da enum birden fazla protokolü destekleyebilir. Bunların bildirimdeki yazım sırasının hiçbir önemi yoktur. Örneğin:

```
protocol A {
    func foo()
}
```



```

protocol B {
    func bar()
}

class Sample : A, B {
    func foo()
    {
        //...
    }

    func bar()
    {
        //...
    }
    //...
}

```

Eğer bir sınıf hem başka bir sınıftan türetilmişse hem de birtakım protokolleri de destekliyorsa sınıfın taban listesinde (yani ':' atomuyla belirtilen listede) önce sınıf isminin belirtilmesi zorunludur. Örneğin:

```

protocol A {
    func foo()
}

protocol B {
    func bar()
}

class Sample {
    //...
}

class Mample : Sample, A, B {
    func foo()
    {
        //...
    }

    func bar()
    {
        //...
    }
    //...
}

```

Bir protokol depolanmış property içeremez ancak hesaplanmış property içerebilir. Protokol içerisinde hesaplanmış property bildirimi yapılırken get ve set bölümlerinin gövdesi yazılmaz. Yalnızca get ve set anahtar sözcükleri bulundurulur. Örneğin:

```

protocol A {
    func foo()
    var count: Int {get set}
}

class Sample : A {
    var val: Int
}

```

```

init()
{
    val = 0
}

func foo()
{
    //...
}

var count: Int {
    get {
        return val
    }

    set {
        val = newValue
    }
}

let s = Sample()
s.val = 10
print(s.val)

```

Protokoldeki bir hesaplanmış property onu destekleyen türlerde depolanmış property biçiminde bildirilebilir. Örneğin:

```

protocol A {
    func foo()
    var count: Int {get set}
}

class Sample : A {
    var count: Int = 0

    func foo()
    {
        //...
    }
}

let s = Sample()
s.count = 10
print(s.count)

```

Protokollerdeki property'ler read-only olabilir. Yani yalnızca get bölümüne sahip olabilir. Bu durumda o protokol desteklenirken o property read-only olarak ya da read/write olarak bildirilebilir. Örneğin:

```

protocol A {
    func foo()
    var count: Int { get }
}

class Sample : A {
    var a: Int

```

```

init(_ a: Int)
{
    self.a = a
}

func foo()
{
    //...
}

var count: Int {
    get {
        return a
    }
    set {
        a = newValue
    }
}
}

let s = Sample(10)
s.count = 10
print(s.count)

```

Hesaplanmış property bildiriminde let anahtar sözcüğünün kullanılmadığını anımsayınız. Ancak read-only property içeren bir protokol depolanmış property ile desteklenirken let anahtar sözcüğü kullanılabilir. Örneğin:

```

protocol A {
    func foo()
    var count: Int { get }
}

class Sample : A {
    let count: Int

    init(_ a: Int)
    {
        self.count = a
    }

    func foo()
    {
        //...
    }
}

```

Protokoller static metod içerebilir. Bu durumda o protokolü destekleyen türler o metodu static olarak bulundurmak zorundadır.

```

protocol A {
    func foo()
    var count: Int {get set}
    static func bar() -> Int
}

class Sample : A {

```

```

func foo()
{
    //...
}

var count: Int {
    get {
        return 0
    }

    set {
        //...
    }
}

static func bar() -> Int
{
    return 0
}
}

```

Protokoller Doğuştan Çokbiçimli Mekanizmaya Dahildir

Protokoller türünden referanslar bildirilebilirler. Ancak nesneler yaratılamazlar. Örneğin:

```

protocol A {
    func foo()
    var count: Int {get set}
    static func bar() -> Int
}

var a: A           // geçerli
a = A()            // error!

```

Bir sınıf türünden referans ya da yapı türünden değişken o sınıfın ya da yapının desteklediği protokol referanslarına doğrudan atanabilir. Örneği:

```

protocol A {
    func foo()
}

class Sample : A {
    func foo()
    {
        print("Sample.foo")
    }
}

var a: A
let s = Sample()

a = s    // geçerli

```

Bu durum türemiş sınıftan taban sınıfa referans dönüştürmesine benzetilebilir. Tabii protokoller bu anlamda taban sınıf durumunda değildirler.

Protokoller çokbiçimli mekanizmaya doğuştan dahil durumdadır. Yani bir protokol referansı ile bir protokol metodu ya da property'si kullanıldığında aslında o referansın dinamik türüne ilişkin sınıfın ya da yapının elemanı kullanılmış olur. Örneğin:

```
protocol A {  
    func foo()  
}  
  
class Sample : A {  
    func foo()  
    {  
        print("Sample.foo")  
    }  
}  
  
var a: A  
let s = Sample()  
  
a = s // geçerli  
a.foo()
```

Örneğin:

Protokoller sayesinde aynı sınıftan türetilmemiş sınıflar ile çokbiçimli işlemler yapılabilir. Aynı zamanda belli bir sınıf değişik protokolleri destekleyerek farklı konularda çokbiçimli olarak kullanılabilir. Örneğin:

```
protocol P {  
    func operate(_ a: Int, _ b: Int) -> Int  
}  
  
class Add : P {  
    func operate(_ a: Int, _ b: Int) -> Int  
    {  
        return a + b  
    }  
}  
  
class Sub : P {  
    func operate(_ a: Int, _ b: Int) -> Int  
    {  
        return a - b  
    }  
}  
  
class Multiply : P {  
    func operate(_ a: Int, _ b: Int) -> Int  
    {  
        return a * b  
    }  
}  
  
class Divide : P  
{  
    func operate(_ a: Int, _ b: Int) -> Int  
    {
```

```

        return a / b
    }
}

func test(_ p: P, _ a: Int, _ b: Int)
{
    let result = p.operate(a, b)
    print(result)
}

let a = Add()
let b = Sub()
let c = Multiply()
let d = Divide()

test(a, 20, 10)
test(b, 20, 10)
test(c, 20, 10)
test(d, 20, 10)

```

Protokollerde çokbiçimliliğin doğal bir durum olduğuna ve override gibi bir anahtar sözcüğün kullanılmadığına dikkat ediniz.

Taban sınıf bir protokolü destekliyorsa türemiş sınıfın da o protokolü desteklediği kabul edilir. Dolayısıyla biz türemiş sınıf türünden bir referansı taban sınıfın desteklediği protokol referansına doğrudan atayabiliriz. Örneğin:

```

protocol P {
    func foo()
}

class A : P {
    func foo()
    {
        print("A.foo")
    }
}

class B : A {
    //...
}

var p: P
let b = B()

p = b          // geçerli
p.foo()        // A.foo çağrılır

```

Tabii türemiş sınıf taban sınıfın metodunu override edebilir. Bu durumda türemiş sınıftaki metot da çokbiçimli mekanizmaya dahil edilmiş olur. Örneğin:

```

protocol P {
    func foo()
}

```

```

class A : P {
    func foo()
    {
        print("A.foo")
    }
}

class B : A {
    override func foo()
    {
        print("B.foo")
    }
}

var p: P
let b = B()

p = b          // geçerli
p.foo()        // B.foo çağrılır

```

Türemiş sınıf bir protokolü destekliyor olsun. Bu protokolün elemanlarının bir kısmı taban sınıfta zaten bildirilmişse türemiş sınıf yalnızca geri kalan elemanları bildirebilir. Örneğin:

```

protocol P {
    func foo()
    func bar()
}

class A {
    func foo()
    {
        print("A.foo")
    }
}

class B : A, P {          // geçerli
    func bar()
    {
        print("B.bar")
    }
}

let p: P = B()
p.foo()
p.bar()

```

Bir protokol türünden bir collection bir nesne söz konusu olabilir. Bu durumda o nesnenin içerisine biz o protokolü destekleyen herhangi bir sınıf, yapı ya da enum nesnelerini yerleştirebiliriz. Sonra heterojen collection nesnemizi dolaşarak protokol metodlarını çağırabiliriz. Örneğin:

```

protocol P {
    func foo()
}

class A : P {
    func foo()
    {

```

```

        print("A.foo")
    }
    //...
}

class B : P {
    func foo()
    {
        print("B.foo")
    }
    //...
}

struct C : P {
    func foo()
    {
        print("C.foo")
    }
    //...
}

var ps: [P] = [A(), B(), A(), C(), B()]

for p in ps {
    p.foo()
}

```

Protokollere Neden Gereksinim Duyulmaktadır?

Swift'te (Java ve C#'ta da böyle) protokollere duyulan gereksinim üç maddeyle açıklanabilir:

- 1) Protokoller Swift'te çoklu türetmenin olmamasının yarattığı boşluğu kısmen doldurmaktadır. Böylece bir sınıf ya da yapı farklı amaçlarla oluşturulmuş protokolleri destekleyerek farklı konulara ilişkin çokbiçimli davranışlar gösterebilmektedir.
- 2) Protokoller bir kontrat görevini de yerine getirirler. Yani bir sınıf, yapı ya da enum bir protokolü destekliyorsa kesinlikle o protokolün elemanlarını bulundurmak zorundadır.
- 3) Yapılar ve enum'lar türetmeye kapalıdır. Ancak protokoller sayesinde onlar da çokbiçimli mekanizmaya dahil edilmişlerdir.

Protokollerde mutating Metotlar

Bir protokoldeki metot mutating anahtar sözcüğü ile bildirilirse o protokolü destekleyen yapılar ve enum'lar o metodu mutating olarak bildirmek zorundadır. Ancak o protokolü destekleyen sınıflar için mutating belirleyicinin bir anlamı yoktur. Örneğin:

```

protocol Test {
    mutating func foo()
}

struct Sample : Test {
    var a: Int
}

```



```

mutating func foo()    // geçerli
{
    a = 10
}

}

class Mample : Test {  // geçerli
    func foo()
    {
        //...
    }
}

```

Ayrıca protokollerdeki normal metotlar yapılarda ve enum'larda mutating olarak bildirilemez.

Protokoller init metotlarına sahip olabilirler. Bu durumda o protokolü destekleyen sınıflarda ya da yapılarda o init metotlarının aynı biçimde gerekir. Ayrıca protokolü destekleyen sınıflardaki init metotlarının başına da required anahtar sözcüğünün getirilmesi zorunludur. Örneğin:

```

protocol P {
    init(_ a: Int)
    func foo()
    func bar()
}

class A : P {
    func foo()
    {
        //...
    }
    func bar()
    {
        //...
    }
    required init(_ a: Int)    // required anahtar sözcüğü zorunlu
    {
        //...
    }
    //...
}

struct B : P {
    func foo()
    {
        //...
    }
    func bar()
    {
        //...
    }
    init(_ a: Int)    // required anahtar sözcüğü zorunlu
    {
        //...
    }
    //...
}

```

required belirleyicisi yapılarda ve enum'larda kullanılmadığı için init içeren protokolü bir yapı ya da enum destekliyse onların init bildirimlerinin önünde required belirleyicisi bulunmaz.

Tabii protokoldeki init metotları her zaman "designated" init metodu biçimindedir. Bunların önüne "convenience" anahtar sözcüğü getirilemez.

Protokollerde failable init metotları da bulunabilir. Bu init metotları o protokolü destekleyen sınıf ya da yapılarda failable olarak ya da normal olarak bildirilebilir. Örneğin:

```
protocol P {
    init?()
    func foo()
    func bar()
}

class A : P {
    required init?()
    {
        //...
    }

    func foo()
    {
        //...
    }

    func bar()
    {
        //...
    }
    //...
}
```

Ancak bu durumun tersi geçerli değildir. Yani protokoldeki init metodu normal ise biz onu destekleyen sınıf ya da failable olarak bildiremeyiz.

Eklentilerde Protokol Desteğinin Belirtilmesi

Bir sınıf ya da yapı için yazılan eklentiler o sınıf ya da yapıya protokol desteği verebilirler. Örneğin:

```
protocol P {
    func foo()
}

class A {
    init()
    {
        //...
    }
    //...
}

//...

extension A : P {
```

```

func foo()
{
    //...
}

```

Tabii protokoldeki elemanlar zaten asıl sınıfta varsa bu durum yine anlamlıdır. Örneğin:

```

protocol P {
    func foo()
}

class A {
    init()
    {
        //...
    }

    func foo()
    {
        //...
    }
    //...
}
//...
extension A : P {

}

let p: P = A()    // geçerli

```

Yalnızca Sınıfların Destekleyebildiği Protokoller

Swift'te (C#'ta bu özellik yok) bir protokolün yalnızca sınıflar tarafından desteklenmesi sağlanabilir. Bunun için protocol bildiriminde ':' atomundan sonra class anahtar sözcüğü yerleştirilir. Böyle protokolleri yapılar ve enum'lar destekleyemezler. Örneğin:

```

protocol P : class {
    func foo()
}

class A : P {    // geçerli
    func foo()
    {
        print("A.foo")
    }
    //...
}

struct C : P {    // error
    func foo()
    {
        print("C.foo")
    }
    //...
}

```

Fakat bir protokolün yalnızca yapılar ve/veya enum'lar tarafından desteklenmesi gibi bir durum yoktur.

Protokollerin Birleştirilmesi (Protocol Composition)

Birden fazla protokol ile birleştirilmiş referanslar bildirilebilir. Bu durumda o referansa ancak o bildirimde belirtilen protokollerin hepsini destekleyen bir sınıf, yapı ya da enum değişkeni atanabilir. Bildirimde protokol birleştirme işlemi protokol isimlerinin arasına & atomu getirilerek yapılır. Örneğin:

```
protocol P1 {
    func foo()
}

protocol P2 {
    func bar()
}

func test(_ p: P1 & P2)
{
    p.foo()
    p.bar()
}

class Sample : P1, P2 {
    func foo()
    {
        print("Sample.foo")
    }

    func bar()
    {
        print("Sample.bar")
    }
}

let s = Sample()
test(s)
```

Tabii protokol birleştirmesinde referans bildirilirken yazılan protokollerin sırasının bir önemi yoktur. Örneğin:

```
var pa: P1 & P2
var pb: P2 & P1
```

Protokollerde Türetme

Bir protokol başka bir protokolden türetilebilir. (Burada "türetme" terimi kullanılmaktadır. "Destekleme" terimi kullanılmamaktadır.) Bu durumda türemiş protokol sanki taban protokolün de elemanlarını içeriyormuş gibi bir etki söz konusu olur. Yani türemiş protokolü destekleyen bir sınıf, yapı ya da enum hem taban sınıfın hem de türemiş sınıfın elemanlarını bildirmek zorundadır. Örneğin:

```
protocol P1 {
    func foo()
}
```

```

protocol P2 : P1 {
    func bar()
}

class Sample : P2 {
    func foo()
    {
        print("Sample.foo")
    }

    func bar()
    {
        print("Sample.bar")
    }
}

```

Tıpkı C#'ta olduğu gibi sınıf, yapı ya da enum'un taban listesinde hem türemiş arayüzün hem de taban arayüzün ismi bulundurulabilir. Bunun hiçbir özel anlamı yoktur. Bazı programcılar okunabilirliği artırmak için bunu tercih edebilmektedir. Yani örneğin:

```

protocol P1 {
    func foo()
}

protocol P2 : P1 {
    func bar()
}

class Sample : P2 {
    //...
}

```

ile:

```

protocol P1 {
    func foo()
}

protocol P2 : P1 {
    func bar()
}

class Sample : P2, P1 {
    //...
}

```

tamamen eşdeğerdir.

Türemiş protokol referansı onun tüm taban protokol türlerine ilişkin referanslara doğrudan atanabilir. Örneğin:

```

let s = Sample()
let p2: P2 = s
let p1: P1 = p2    // geçerli, türemiş tabana atama

```

Protokollerde is ve as Operatörlerinin Kullanımı

Bir protokol referansı ile biz is operatörünü kullanabiliriz. Bu durumda bu protokol referansının dinamik türünün ilgili sınıf ya da yapıyı içerip içermediğine bakılır.. Örneğin:

```
protocol P {
    func foo()
}

class A : P {
    func foo()
    {
        print("A.foo")
    }
}

class B : A {
    //...
}

class C : P {
    func foo()
    {
        print("C.foo")
    }
}

func test(_ p: P)
{
    if p is A {
        print("p'nin dinamik türü A'yı içeriyor")
    }
    if p is B {
        print("p'nin dinamik türü B'yi içeriyor")
    }

    if p is C {
        print("p'nin dinamik türü C'yi içeriyor")
    }
    print("-----")
}

let a = A()
let b = B()
let c = C()

test(a)
test(b)
test(c)
```

Biz bir referansın dinamik türünün bir protokolü destekleyip desteklemediğini de is operatörü ile sorgulayabiliriz. Örneğin:

```
protocol P1 {
    func foo()
}

protocol P2 {
```

```

    func bar()
}

class A : P1, P2 {
    func foo()
    {
        print("A.foo")
    }

    func bar()
    {
        print("A.bar")
    }
}

let p1: P1 = A()

if p1 is P2 {
    print("Ok")
}
else {
    print("not ok")
}

```

Protokoller türünden referanslar da as? ve as! operatörlerinin sol taraf operandı olarak kullanılabilir. Başka bir deyişle biz bir protokol türünden referansı bir sınıf, yapı ya da enum türüne asışığıya doğru dönüştürme işlemine (downcasting) sokabiliriz. Örneğin:

```

protocol P {
    func foo()
}

class Sample : P {
    var a: Int

    init(a: Int)
    {
        self.a = a
    }

    func foo()
    {
        print("A.foo")
    }
}

var p: P
var s: Sample

p = Sample(a: 10)           // upcast
s = p as! Sample           // downcast
print(s.a)

```

as! ve as? operatörlerinin sol tarafındaki protokol referansına ilişkin protokol türü sağ tarafındaki sınıf türü tarafından desteklenmiyor olsa bile derleme aşamasından başarı ile geçilir. Tabii kontrol yine programın çalışma zamanı sırasında yapılacaktır. Örneğin:

```

protocol P {
    func foo()
}

class Sample {
    //...
}

class Mample : P {
    func foo()
    {
        print("Mample.foo")
    }
    //...
}

var p: P
var s: Sample

p = Mample()

if let k = p as? Sample {
    print("ok")
}
else {
    print("not ok")
}

s = p as! Sample // derleme aşamasından başarıyla geçilir, fakat exception oluşur

```

Peki neden bu durumda denetim derleme aşamasında yapılamamaktadır? Çünkü belki de protokol referansının dinamik türü sağ taraftaki sınıftan türetilmiş bir tür türündendir.

Ancak bir protokol türünden referans as! ya da as? operatörleriyle o protokol tarafından desteklenmeyen yapı ya da enum türüne dönüştürülmek istenirse denetim derleme aşamasında yapılır. Çünkü yapılar enum'lar türetmeye kapalıdır.

Bir sınıf türünden referans as! ya da as? operatörleriyle o sınıfın desteklemediği bir protokol türüne dönüştürülmek istenirse derleme aşamasından yine başarıyla geçilir. Denetim programın çalışma zamanı sırasında yapılır. Örneğin:

```

var s: Sample = Sample()

let p: P? = s as? P

```

Burada Sample sınıfı P protokolünü desteklemiyor olsa bile denetim derleme aşamasında yapılmaz. Programın çalışma zamanı sırasında yapılır. Tabii yine yukarıdakiyle aynı gerekçelerle bir yapı ya da enum türü as! ya da as? operatörleriyle o yapı ya da enum türünün desteklemediği bir protokol türüne dönüştürülmeye çalışılırsa denetim derleme aşamasında yapılır.

Swift'te yine bir protokol türünden referans her zaman aralarında türetme ilişkisi olmasa bile başka bir protokol türüne as! ya da as? operatörleriyle dönüştürülmek istenebilir. Bu durumda denetim derleme aşamasında yapılmaz. Programın çalışma zamanı sırasında yapılır. Örneğin:

```
protocol P1 {
    func foo()
}

protocol P2 {
    func foo()
}

class Sample : P1 {
    func foo()
    {
        //...
    }
    //...
}

let p1: P1 = Sample()
var p2: P2

p2 = p1 as! P2 // derleme aşamasında her zaman başarıyla geçilir, denetim çalışma zamanında yapılır
```

Programın çalışma zamanı sırasında dönüştürülmek istenen protokol referansının (örneğimizdeki p1) dinamik türünün dönüştürülmek istenen protokolü (örneğimizdeki P2) destekleyip desteklemediğine bakılır.

Protokol Eklentileri (Protocol Extensions)

Bir protokol için eklenti oluşturulabilir. Böylelikle protokole yeni elemanlar eklenebilir. Ancak eklentide eklenen elemanların gövde içermesi zorunludur. Örneğin:

```
protocol P {
    func foo()
}

//...

extension P {
    func bar()
    {
        print("extension P.bar")
    }
}

class Sample : P {
    func foo()
    {
        print("Sample.foo")
    }
    //...
}

let s = Sample()
```

```

s.foo()      // geçerli, Sample.foo
s.bar()      // geçerli, extension P.bar

let p: P = s

p.foo()      // geçerli, Sample.foo
p.bar()      // geçerli, extension P.bar

```

Eklentideki elemanlar ilgili sınıf ya da yapıda yeniden bildirilebilir. Bu durumda aşağıdan yani sınıf ya da yapı değişkeni yoluyla bu elemanlar kullanıldığında sınıf ya da yapının içerisinde bildirilmiş elemanların, protokol yoluyla kullanıldığında ise eklentide bildirilmiş olan elemanın kullanıldığı varsayılmaktadır. Örneğin:

```

protocol P {
    func foo()
}

//...

extension P {
    func bar()
    {
        print("extension P.bar")
    }
}

class Sample : P {
    func foo()
    {
        print("Sample.foo")
    }

    func bar()
    {
        print("bar")
    }
    //...
}

let s = Sample()

s.foo()      // geçerli, Sample.foo
s.bar()      // geçerli, Sample.bar

let p: P = s

p.foo()      // geçerli, Sample.foo
p.bar()      // geçerli, extension P.bar

```

Operatör Fonksiyonları (Operator Functions)

Operatör fonksiyonları konusu Java'da yoktur. C#'a C++'tan basitleştirilerek aktarılmıştır. Swift'te de operatör fonksiyonları dahil edilmiştir. Bilindiği gibi operator fonksiyonları dile ekstra bir işlevsellik katmamaktadır. Operatör fonksiyonlarının yalnızca okunabilirliği artırıcı bir işlevi vardır. Bu sayede biz kendi sınıf ya da yapılarımız türünden iki değişkeni sanki onlar temel türlerdenmiş gibi toplama, çıkartma vs. işlemlerine sokabiliriz. Bu

durumda aslında arka planda bizim belirlediğimiz ve ismine "operatör fonksiyonu" denilen fonksiyon çağrılmaktadır.

Swift'te operatör fonksiyonları global düzeyde bildirilmek zorundadır. Operatör fonksiyonları sınıfların ya da yapıların içerisinde bildirilemezler. Bildirimleri sırasında operatör fonksiyonları operatör sembolüyle isimlendirilir. Örneğin:

```
func +(z1: Complex, z2: Complex) -> Complex
{
    //...
}
```

Burada fonksiyonun isminin + operatör sembolünden oluştuğuna dikkat ediniz. Operatör fonksiyonları isimlendirme biçiminin dışında tamamen normal fonksiyonlar gibidir.

Operatör fonksiyonlarının parametreleri ve geri dönüş değerleri herhangi bir türden olabilir. Ancak eğer operatör metodu tek operandlı bir operatöre ilişkinse operatör fonksiyonunun bir parametresi, iki operandlı bir fonksiyona ilişkinse iki parametresi bulunmak zorundadır.

Örneğin:

```
class Complex {
    var real: Double
    var imag: Double

    init(_ real: Double, _ imag: Double)
    {
        self.real = real
        self.imag = imag
    }

    var description: String {
        return self.real.description + " + " + self.imag.description + "i"
    }
}

func +(z1: Complex, z2: Complex) -> Complex
{
    let result = Complex(z1.real + z2.real, z1.imag + z2.imag)

    return result
}

let z1 = Complex(2, 3)
let z2 = Complex(3, 4)
var result: Complex

result = z1 + z2
print(result.description)
```

Swift derleyicisi bir operatörle karşılaştığında önce operand'ların türlerine bakar. Eğer operand'lar temel türlerdence işlemi gerçekleştirir. Fakat operand'lardan en az biri bir sınıf ya da yapı türündense bu işlemi yapabilecek bir operatör fonksiyonu araştırır. Eğer böyle bir fonksiyonu bulursa operand'ları o fonksiyona

argüman olarak gönderir ve fonksiyonu çağırır. Fonksiyonun geri dönüş değeri işlem sonucu olarak elde edilmektedir.

Operatör fonksiyonları kombine edilebilir. Yani birinin sonucu diğerine girdi yapılabilir. Örneğin:

```
result = z1 + z2 + z3
```

Burada z1 ve z2 toplanarak bir değer elde edilmiş, elde edilen değer de z3 ile toplanmıştır. Sonuç da result değişkenine atanmıştır.

Eğer önek bir operatöre ilişkin operatör yazmak istiyorsak operatör bildiriminin başına prefix anahtar sözcüğü, sonek bir operatöre ilişkin operatör metodu yazmak istiyorsak postfix anahtar sözcüğü getirilmelidir. infix operatörler için bildirimin başına hiçbir şey getirilmez. (infix anahtar sözcüğü de getirilmez). Örneğin:

```
import Foundation

struct Number {
    var val: Int

    init()
    {
        val = 0
    }

    init(val: Int)
    {
        self.val = val
    }

    var description: String {
        return val.description
    }
}

func +(a: Number, b: Number) -> Number
{
    var result = Number()

    result.val = a.val + b.val

    return result
}

func -(a: Number, b: Number) -> Number
{
    var result = Number()

    result.val = a.val - b.val

    return result
}

func *(a: Number, b: Number) -> Number
{
    var result = Number()
```

```

    result.val = a.val * b.val

    return result
}

func /(a: Number, b: Number) -> Number
{
    var result = Number()

    result.val = a.val / b.val

    return result
}

prefix func +(a: Number) -> Number
{
    return a
}

prefix func -(a: Number) -> Number
{
    return Number(val: -a.val)
}

let a = Number(val: 10)
let b = Number(val: 10)
let c = Number(val: 40)

let result = -a * +b - c
print(result.description)

```

Swift'te diğer dillerin aksine operatör olmayan atom kümesinden de operatör fonksiyonu oluşturulabilmektedir. Ancak operatör sembollerinin belli karakterlerden seçilmesi zorunlu tutulmuştur. Bu karakterlerin UNICODE tablodaki listesi dilin kendi referans kitabında belirtilmiştir. Bunların görüntüsel karşılığı için <https://gist.github.com/wxs/d773cb2a2e9891dbfd63> adresine başvurulabilir.

Eğer biz operatör olmayan bir sembolden operatör fonksiyonu yazacaksak operatör sembolünü öncelik bakımından nitelendirmemiz gerekir. Nitelendirme işleminin genel biçimi şöyledir:

<operatörün konumu (prefix, infix, postfix)> operator <operatör sembolü> [: <öncelik grubu>]

Operatörün öncelik grubu eskiden sayısal biçimde belirtiliyordu. Daha sonra sayısal belirtmeden vaz geçilerek grupsal belirleme yöntemi izlendi. Öncelik grupları şunlardan oluşmaktadır:

```

BitwiseShiftPrecedence
MultiplicationPrecedence
AdditionPrecedence
RangeFormationPrecedence
CastingPrecedence
NilCoalescingPrecedence
ComparisonPrecedence
LogicalConjunctionPrecedence
TernaryPrecedence
AssignmentPrecedence

```

Swift'te tek operandlı operatörler için öncelik grubu belirtilmemektedir. Örneğin aşağıdaki operatör öncelik bildirimleri geçerlidir:

```
prefix operator Σ
infix operator Σ+: AdditionPrecedence
infix operator Σ+: MultiplicationPrecedence
```

İki operandlı infix operatörlerde de öncelik grubu belirtilmeyebilir. Bu durumda bu operatörler başka operatörlerle kombine edilemezler.

Örneğin:

```
prefix operator Σ

prefix func Σ(a: [Int]) -> Int
{
    var total = 0

    for x in a {
        total += x
    }

    return total
}

let result = Σ[1, 2, 3, 4, 5] + 10

print(result)
```

infix operatör için de şöyle bir örnek düzenlenebilir:

```
infix operator Σ+: AdditionPrecedence

func Σ+(a: [Int], b: [Int]) -> [Int]
{
    var result = [Int]()
    var rest: [Int]
    let minCount = a.count < b.count ? a.count : b.count

    for i in 0..
```

```
var result = [1, 2, 3] Σ+ [5, 7, 9, 10]
print(result)
```

Eskiden Swift'te temel türlere ilişkin operatör fonksiyonları yazılabiliyordu. Sonra bu özellik kaldırıldı. Aşağıda bir rasyonel sayı sınıfının gerçekleştirimini görüyorsunuz:

```
class Rational {
    var a: Int
    var b: Int

    init()
    {
        a = 0
        b = 1
    }

    init(_ a: Int, _ b: Int = 1)
    {
        self.a = a
        self.b = b

        if a != 0 {
            reduce()
        }
    }

    func disp()
    {
        print(a, terminator: "")
        if b != 1 && a != 0 {
            print("/\(b)", terminator: "")
        }
        print()
    }

    func reduce()
    {
        var a = self.a
        var b = self.b
        var temp: Int

        while b != 0 {
            temp = b
            b = a % b
            a = temp
        }

        self.a /= a
        self.b /= a

        if self.b < 0 {
            self.b = -self.b
            self.a = -self.a
        }
    }
}
```

```

func +(_ x: Rational , y: Rational) -> Rational
{
    let result = Rational()

    result.a = x.a * y.b + x.b * y.a
    result.b = x.b * y.b
    result.reduce()

    return result
}

func -(_ x: Rational , y: Rational) -> Rational
{
    let result = Rational()

    result.a = x.a * y.b - x.b * y.a
    result.b = x.b * y.b
    result.reduce()

    return result
}

func *(_ x: Rational , y: Rational) -> Rational
{
    let result = Rational()

    result.a = x.a * y.a
    result.b = x.b * y.b
    result.reduce()

    return result
}

func /(_ x: Rational , y: Rational) -> Rational
{
    let result = Rational()

    result.a = x.a * y.b
    result.b = x.b * y.a
    result.reduce()

    return result
}

prefix func -(_ x: Rational) -> Rational
{
    let result = Rational()

    result.a = -x.a
    result.b = x.b

    return result
}

prefix func +(_ x: Rational) -> Rational
{
    return x
}

```



```
let x = Rational(2, -4), y = Rational(3, 4)
var result: Rational

result = -x + y
result.disp()
```

Sınıflarda ve Yapılarda Erişim Belirleyici Anahtar Sözcükler

Swift'te 2.0 versiyonu ile birlikte private, public ve internal erişim belirleyici anahtar sözcükler eklenmiştir. Yani biz diğer pek çok nesne yönelimli programlama dilinde olduğu gibi sınıfın bazı elemanlarını dışarıdan kullanıma kapatabiliriz. Swift'te internal default erişim belirleyicisidir. Yani erişim belirleyici anahtar sözcüklerden hiçbirisi kullanılmamışsa default olarak internal belirleyicisi kullanılmış gibi etki oluşur. internal bir sınıf ya da yapı elemanına aynı modülden (başka bir kaynak dosya da dahil olmak üzere) erişilebilir. Ancak başka bir modülde bulunan bir sınıf ya da yapının internal elemanlarına erişilemez. Yani internal belirleyicisi ile gizleme modül temelinde etki göstermektedir. private belirleyicisi yalnızca ilgili elemana aynı kaynak dosyadan erişilebileceğini belirtmektedir. public belirleyicisi ise elemana her yerden erişilebileceği anlamına gelmektedir. Yani örneğin bir modül içerisinde üç kaynak dosya bulunuyor olsun. Eğer eleman private ise yalnızca aynı modül söz konusu olsa bile aynı kaynak dosyadan o elemana erişilebilir. Eğer eleman public erişim belirleyicisi ile bildirilmişse elemana her modülden yani her yerden erişilebilir. Swift'te C++, Java ve C#'ta olduğu gibi protected belirleyicisi yoktur. Örneğin:

```
public class Sample {
    private var a: Int

    public init()
    {
        a = 0
    }

    public func foo()
    {
        //...
    }

    private func bar()
    {
        //...
    }

    internal func tar()
    {
        //...
    }
    //...
}

var s: Sample

s = Sample()           // geçerli, init public
s.tar()                // geçerli, tar internal

s.a = 10               // error! a private
s.bar()                // error! bar private
```

Daha sonra Swift'te fileprivate biçiminde yeni bir erişim belirleyicisi daha eklenmiştir. Bu erişim belirleyicisi ilgili elemanın yalnızca belli bir kaynak dosyadan (kaynak dosyalar modülleri oluşturmaktadır) erişilmesin, sağlamaktadır. Bu durumda erişim belirleyicileri gvwşekten sıkıya doğru şöyle sıralanmaktadır:

```
public
internal
fileprivate
private
```

Ayrıca Swift'te türlerin başına da erişim belirleyici anahtar sözcükler getirilebilmektedir. Eğer bir tür dışarıda bildirilmişse onun başına public, internal ya da fileprivate belirleyicisi getirilebilir. Burada da default durum internal'dır. internal bir sınıf yalnızca kendi modülünden kullanılabilir. Biz bir modül yazarken ilgili sınıfın başka bir modül tarafından kullanılmasını istiyorsak sınıf bildiriminin başına public belirleyicisini getirmeliyiz. Örneğin:

```
public class Sample {
    //...
}
```

Tabii public olmayan bir sınıfın public elemana sahip olması anlamlı değildir. Swift derleyici bu durumda uyarı mesajı vermektedir.

guard Deyimi

guard deyimi yalnızca else bölümü bulunan if deyimine benzetilebilir. Genel biçimi şöyledir:

```
guard <Bool türden ifade> else {
    //...
}
```

guard deyiminin yalnızca else kısmı vardır. Bu kısım kontrol ifadesi false ise çalıştırılır. guard deyimi bir çeşit assert etkisi yaratmak için dile sokulmuştur. Yani bu deyimde programcı sağlanması gereken koşulu belirtir. Bu koşul sağlanmıyorsa deyimden else kısmı çalışır. guard deyiminin else kısmından normal akışsal çıkış yasaklanmıştır. Programcının else bloğu içerisinde akışı başka bir yere yöneltmesi zorunludur. Bu da tipik olarak return gibi, break gibi continue gibi deyimlerle ya da exception fırlatan throw işlemiyle yapılabilir. Örneğin:

```
func foo(_ a: Int)
{
    guard a >= 0 else {
        print("parameter cannot be negative")
        return
    }
    print("ok")
}

foo(10)
foo(-20)
```

Görüldüğü gibi burada guard deyiminin else kısmında akışın else bloğunu bitirmesine izin verilmemiştir. Yani guard deyiminden sonraki deyimler ancak koşul doğruysa çalıştırılmaktadır. Biz bir guard deyimi gördüğümüzde şunları düşünmeliyiz: "Akış guard deyimi aşıp aşağıya düşmüşse demek ki guard deyiminde belirtilen koşul

sağlanmıştır. Eğer guard deyiminde belirtilen koşul sağlanmamış olsaydı guard deyiminin else kısmı çalıştırılır ve akışın aşağıya düşmesine izin verilmezdi.

guard deyimi daha çok seçeneksel türler için tercih edilmektedir. Tıpkı if deyiminde olduğu gibi guard deyiminin de let ya da var ile oluşturulan seçeneksel türlerle çalışan biçimi vardır. guard deyiminin bu biçiminde yine nil karşılaşt. Örneğin:ırması ve otomatik unwrap işlemi uygulanmaktadır. Örneğin:

```
func foo(_ a: Int?)
{
    guard let b = a else {
        print("parameter cannot be nil")
        return
    }
    print("\(b)")        // geçerli
}

foo(10)
foo(nil)
```

guard deyiminin let/var ile kullanılan seçeneksel biçiminin if deyiminin let/var ile seçeneksel biçiminden önemli bir farkı vardır. guard deyiminin seçeneksel biçiminde bildirilen değişken guard deyimi dışında da kullanılabilir. Ancak if deyiminin seçeneksel biçiminde bildirilen değişken if deyiminin dışında kullanılamaz. Örneğin:

```
func foo(_ a: Int?)
{
    if let b = a { }
    else {
        print("parameter cannot be nil")
        return
    }
    print("\(b)")        // error
}

foo(10)
foo(nil)
```

Burada if deyiminin dışında b'ye erişilemediğine dikkat ediniz. Örneğin:

```
var dict: [String: Int] = ["Ali": 123, "Veli": 234, "Selami": 543]

func foo(_ name: String)
{
    guard let val = dict[name] else {
        print("name cannot find")
        return
    }
    print(val)
}

foo("Ali")
foo("Sacit")
```

defer Deyimi

defer deyiminin genel biçimi şöyledir:

```
defer {  
    //...  
}
```

defer deyimi bir bloğun içerisindeyse o bloktan çıkıldığında çalıştırılır, akış defer deyimine geldiğinde çalıştırılmaz. Aynı blok içerisinde birden fazla defer varsa bunlar bloktan çıkış sırasında ters sırada çalıştırılmaktadır. Örneğin:

```
func foo(_ a: Int?)  
{  
    print("one")  
    defer {  
        print("defer 1")  
    }  
    print("two")  
    defer {  
        print("defer 2")  
    }  
    print("three")  
}  
  
foo(10)
```

Burada ekrana şunlar basılacaktır:

```
one  
two  
three  
defer 2  
defer 1
```

defer bloktan nasıl çıkılırsa çıkılsın çalıştırılır. Yani normal çıkış, return ile çıkış, break ve continue ile çıkışlarda ya da exception nedeniyle çıkışlarda da defer deyimleri çalıştırılır. Örneğin:

```
func foo(_ a: Int)  
{  
    defer {  
        print("defer 1")  
    }  
    defer {  
        print("defer 2")  
    }  
    if a < 0 {  
        return  
    }  
    print("one")  
    print("two")  
    print("three")  
}  
  
foo(-10)
```

Pekiye defer deyiminin kullanım nedeni nedir? İşte blok içerisinde birtakım kaynaklar tahsis edilmiş olabilir. Daha sonra bir exception oluşabilir. Bu durumda defer deyiminde tahsisatlar geri bırakılabilir. Örneğin bir kaynağın tipik kullanımı aşağıdaki gibi olsun:

```
func foo()
{
    <kaynak tahsis et>

    <kaynakla işlem yap>

    <kaynağı bırak>
}
```

Buradaki problem kaynak tahsis edildikten sonra o kaynak kullanılırken oluşan exception ile akışın başka bir yere atlayabilmesidir. İşte exception oluştuğunda kaynağın otomatik boşaltımı defer sayesinde şöyle yapılabilir:

```
func foo()
{
    <kaynak tahsis et>
    defer {
        <kaynağı bırak>
    }
    <kaynakla işlem yap>
}
```

Bu tür işlemler Java ve C#'ta try-finally bloklarıyla ya da C#'taki using deyimleriyle gerçekleştirilebilmektedir. C++'ta "stack unwinding" mekanizması olduğu için zaten bu tür sorunlar sınıfların destructor metotlarıyla çözülmektedir.

defer deyiminin blok çıkışında çalıştırılabilmesi için akışın defer üzerinden geçmiş olması gerekir. Örneğin:

```
func foo(a: Int)
{
    defer {
        print("defer 1")
    }
    if a < 0 {
        return
    }

    defer {
        print("defer 2")
    }
}
```

Burada a eğer sıfırdan küçükse ikinci defer bloktan çıkıldığında çalıştırılmayacaktır. Ancak o ana kadar akışın ulaştığı defer'ler bloktan çıkılırken çalıştırılır.

Exception İşlemleri

Exception konusu Swift'e 2.0 versiyonuyla sokulmuştur. Swift'in exception mekanizması C++, Java ve C#'tan biraz farklıdır. Swift'te bir türün exception amacıyla fırlatılabilmesi için onun Error isimli boş bir protokolü destekliyor olması gerekir. Örneğin:

```

class MyException : Error {
    //....
}

enum YourException : Error {
    //...
}

```

Swift'te enum'lar exception mekanizmasında çok daha yaygın olarak kullanılmaktadır. Çünkü enum'ların her case bölümü bir exception cinsini belirtebilmektedir. Örneğin:

```

enum MyException : ErrorType {
    case NotFound
    case IncorrectArgument
    case OutOfRange
}

```

Exception'ın throw edilmesinde yine throw deyimi kullanılır. Örneğin:

```

throw MyException.NotFound

```

Swift'te fonksiyon ya da metotlarda "exception belirlemesi" vardır. Eğer bir fonksiyon ya da metot dışına bir exception throw edilmişse bu durum fonksiyon ya da metodun bildiriminde parametre parantezinden sonra throws anahtar sözcüğüyle belirtilmek zorundadır. Örneğin:

```

func foo() throws
{
    //...
}

```

Fakat Swift'te Java'daki gibi fonksiyonun ya da metodun hangi tür ile throw ettiği belirtilmemektedir. Eğer fonksiyon ya da metodun geri dönüş değeri varsa -> atomu throws anahtar sözcüğünden sonraya yerleştirilir. Örneğin:

```

func mysqrt(_ a: Double) throws -> Double
{
    if a < 0 {
        throw MyException.IncorrectArgument
    }
    return sqrt(a)
}

```

Swift'te C++, Java ve C#'ta olduğu gibi try bloğu yoktur. Onun yerine do bloğu vardır. Örneğin:

```

try foo()

```

gibi. Exception'ı yakalama do-catch bloklarıyla yapılır.

```

do {
    //...
}

```

```

catch MyException.NotFound {
    //...
}

catch MyException.IncorrectArgument {
    //...
}

catch MyException.OutOfRange {
    //...
}

```

Akış do bloğu içerisinde girdiğinde artık exception kontrolü uygulanmaktadır. (Başka bir deyişle buradaki do, diğer dillerdeki try gibidir.) Bu durumda eğer do bloğunda bir exception oluşursa akış do bloğunun aynı türden catch bloğuna aktarılır. O catch bloğu çalıştırdıktan sonra diğer catch blokları atlanır ve akış catch bloklarının sonundan devam eder. Örneğin:

```

import Foundation

enum MyException : Error {
    case NotFound
    case IncorrectArgument
    case OutOfRange
}

func mysqrt(_ a: Double) throws -> Double
{
    if a < 0 {
        throw MyException.IncorrectArgument
    }
    return sqrt(a)
}

var result: Double

print("Bir sayı giriniz:", terminator: "")

do {
    if let val = Double(readLine()!) {
        result = try mysqrt(val)
        print(result)
    }
    else {
        print("geçersiz sayı!..")
    }
}

catch MyException.IncorrectArgument {
    print("Negatif sayı kullanılamaz!")
}

```

Akış do bloğuna girdikten sonra hiç exception oluşmazsa catch blokları atlanır ve akış catch bloklarının sonundan devam eder.

Eğer bir fonksiyon throws ile exception fırlatacağını belirtmişse onun kesinlikle try ile nitelendirilmesi zorunludur. Fakat try deyiminin do bloğu içerisinde bulundurulması zorunlu tutulmamıştır. Ancak yakalanamayan exception'lar yine Java ve C#'ta olduğu gibi programın çökmesine yol açarlar.

Örneğin:

```
enum MyException : Error {
    case NotFound
    case IncorrectArgument
    case OutOfRange
}

func search(_ name: String) throws -> Int
{
    var dict: [String: Int] = ["Ali": 123, "Veli": 234, "Selami": 543]

    guard let val = dict[name] else {
        throw MyException.NotFound
    }

    return val
}

while true {
    do {
        print("Bir isim giriniz:", terminator:"")
        let name = readLine()!
        if name == "exit" {
            break
        }
        let no = try search(name)

        print(no)
    }
    catch MyException.NotFound {
        print("isim bulunamadı!")
    }
}
```

catch bloğunun genel biçimi şöyledir:

```
catch [tür] [(bildirim)] [kalıp] {
    //...
}
```

Eğer catch anahtar sözcüğünün yanı boş bırakılırsa bu durum her türden exception'ın yakalanacağı anlamına gelir.

throw işlemi ile bazı bilgiler de exception'ı yakalayacak catch cümlesine gönderilebilir. Bu örneğin enum elemanlarına tür bilgisi atayarak gerçekleştirilebilir. Örneğin:

```
import Foundation

enum MyException : Error {
    case NotFound
    case IncorrectArgument
    case OutOfRange(String)
}
```



```

func mysqrt(_ a: Double) throws -> Double
{
    guard a >= 0 else {
        throw MyException.OutOfRange("value must be positive or zero!")
    }

    return sqrt(a)
}

do {
    let result = try mysqrt(-20)
    print(result)
}

catch MyException.OutOfRange(let msg) {
    print("Error: \(msg)")
}

```

Burada catch sentaksına dikkat ediniz: catch parametresinde tür belirtilmemektedir. Yani bildirim aşağıdaki gibi yapılmamalıdır:

```

catch MyException.OutOfRange(var msg: String) {    // error!
    print("Error: \(msg)")
}

```

Tabii enum elemanları bir sınıf ya da yapı türünden de olabilir. Örneğin:

```

import Foundation

enum MyException : Error {
    case NotFound
    case IncorrectArgument(IncorrectArgumentException)
    case OutOfRange
}

class IncorrectArgumentException {
    var msg: String

    init(msg: String)
    {
        self.msg = msg;
    }

    var description : String {
        return msg
    }
}

func mysqrt(_ a: Double) throws -> Double
{
    guard a >= 0 else {
        throw MyException.IncorrectArgument(IncorrectArgumentException(msg: "value must be
positive or zero!"))
    }

    return sqrt(a)
}

```

```
do {
    let result = try mysqrt(-20)
    print(result)
}

catch MyException.IncorrectArgument(let e) {
    print("Error: \(e.description)")
}
```

Aslında enum'ların dışında sınıflar ve yapılarla da throw işlemi yapılabilir. Fakat bunların yakalanması için catch cümlesinde where kalıbının kullanılması gerekir. Şöyle ki: Aslında catch cümlesinden biz Error trefüründen reransı elde ederiz. Bu referansın dinamik türüne ilişkin bir kalıp oluşturarak catch düzenlemesini yapabiliriz. Örneğin:

```
import Foundation

class IncorrectArgumentException : Error {
    var msg: String

    init(msg: String)
    {
        self.msg = msg;
    }

    var description : String {
        return msg
    }
}

func mysqrt(_ a: Double) throws -> Double
{
    guard a >= 0 else {
        throw IncorrectArgumentException(msg: "value must be positive or zero!")
    }

    return sqrt(a)
}

do {
    let result = try mysqrt(-20)
    print(result)
}

catch let e where e is IncorrectArgumentException {
    let iae = e as! IncorrectArgumentException
    print("\(iae.description)")
}
```

Tabii yukarıda da belirtildiği gibi Swift'in exception mekanizması temelde enum'larla kullanılmak üzere tasarlanmıştır. Yukarıdaki gibi doğrudan bir sınıf ile throw etmek nadir rastlanabilecek bir tekniktir.

try operatörünün yanı sıra exception kontrolü için try? ve try! operatörleri de vardır. try? ile exception kontrolü uygulandığında eğer try? operatörünün yanındaki ifadede exception oluşursa akış catch bloğuna

aktarılmamaktadır. Bunun yerine bu operatör nil değerini üretmektedir. Tabii bu durumda try? operatörünün de bloğu içerisinde kullanılmasının da bir anlamı yoktur. Örneğin:

```
import Foundation

enum MyException : Error {
    case IncorrectArgument(String)
}

func mysqrt(_ a: Double) throws -> Double
{
    guard a >= 0 else {
        throw MyException.IncorrectArgument("value must be positive or zero!")
    }

    return sqrt(a)
}

let result: Double? = try? mysqrt(-20)

if let r = result {
    print("\(r)")
}
else {
    print("exception occurred!")
}
```

Tabii try? operatörünün kullanılabilmesi için bu operatörün sağındaki ifadenin bir değer veriyor olması gerekir. Yani örneğin foo fonksiyonu bir değer geri döndürmeseydi biz try? operatörünü kullanamazdık.

try! operatöründe eğer exception oluşursa programın çalışma zamanı sırasında program çöker. Exception oluşmazsa akış normal biçimde devam etmektedir. try! operatörünün de do bloğunun içerisinde kullanılmasının bir anlamı yoktur. Örneğin:

```
import Foundation

enum MyException : Error {
    case IncorrectArgument(String)
}

func mysqrt(_ a: Double) throws -> Double
{
    guard a >= 0 else {
        throw MyException.IncorrectArgument("value must be positive or zero!")
    }

    return sqrt(a)
}

let result: Double? = try! mysqrt(-20)
try! operatörü exception oluşması durumunda programın çökmesine yol açtığı için bir değer elde edilmesine yol açmamaktadır. Örneğin:

x = try! foo()
```

Burada foo'da exception oluşursa zaten program çöker. Bu nedenle try! operatörü try? operatörü gibi seçeneysel bir tür vermemektedir.

Generic'ler

Yeni dillerin pek çoğunda olduğu gibi Swift'te de generic sınıflar ve fonksiyonlar vardır. Generic'ler konusu Java ve C#'takine sentaks olarak oldukça benzemektedir. Ancak Swift'teki generic mekanizma işleyiş bakımından daha çok C++'a benzetilebilir.

Bazen farklı türler için aynı işi yapan birden fazla fonksiyonun ya da metodun yazılması gerekebilmektedir. Örneğin:

```
func getMax(_ a: [Int]) -> Int
{
    var max = a[0]

    for x in a {
        if max < x {
            max = x
        }
    }

    return max
}

let a = [4, 67, 34, 23, 6]
let max = getMax(a)
print(max)
```

Yukarıdaki getMax fonksiyonu Int bir dizinin en büyük elemanına geri dönmektedir. Ancak biz Double bir dizinin en büyük elemanını elde etmek istersek içi tamamen yukarıdaki gibi olan fakat parametre ve geri dönüş değerinde Double kullanılan yeni bir getMax fonksiyonunu yazmak zorundayız:

```
func getMax(_ a: [Double]) -> Int
{
    var max = a[0]

    for x in a {
        if max < x {
            max = x
        }
    }

    return max
}
```

İşte generic özelliği yukarıdaki gibi içi aynı olan fakat farklı türler için tekrar tekrar yazılması gereken fonksiyonlar ve sınıfların daha pratik oluşturulması için düşünülmüştür. Böylece generic fonksiyonlar, sınıflar ve yapılar bir ya da birden fazla türe dayalı olarak bir şablon biçiminde bildirilebilirler. Derleyici de o şablona bakarak ilgili türden fonksiyonu, sınıfı ya da yapıyı bizim için oluşturur.

Generic Fonksiyonlar ve Metotlar

Generic bir fonksiyonun bildirimi tıpkı C# ve Java'da olduğu gibi fonksiyon ya da metot isminden sonra açılabilir parantezlerle tür parametrelerinin bildirilmesiyle yapılır. Örneğin:

```
func foo<T, K>(a: T, b: K) -> T
{
    //...
}
```

Burada generic fonksiyonun tür parametreleri T ve K'dır. Bu T ve K herhangi iki türü temsil etmektedir. Tür parametreleri geleneksel olarak pascal yazım tarzıyla (yani ilk harfi büyük diğerleri küçük olacak biçimde) harflendirilmektedir. T ve K gibi tek harfli isimler de çok tercih edilmektedir. Örneğin swap fonksiyonu generic olarak şöyle yazılabilir:

```
func swap<T>( a: inout T, _ b: inout T)
{
    let temp = a
    a = b
    b = temp
}

var a: Int = 10, b: Int = 20

swap(&a, &b)
print("a = \(a), b = \(b)")

var x: Double = 12.3, y: Double = 20.5
swap(&a, &b)
print("x = \(a), y = \(b)")
```

Buradaki T türü generic fonksiyon kullanılırken derleyici tarafından argümanlara bakılarak otomatik tespit edilir. Burada swap iki Int argümanla çağırıldığı için derleyici T türünün Int olduğunu tespit eder. Swift'te C++ ve C#'ta olduğu gibi generic türlerin açıkça belirtilmesi özelliği yoktur. Bu nedenle Swift'te tüm generic tür parametrelerinin fonksiyon ya da metodun imzası içerisinde kullanılıyor olması zorunludur. Yani örneğin aşağıdaki generic bildirim C++ ya da C# karşılığı dikkate alındığında geçerli olduğu halde Swift'te geçerli değildir:

```
func foo<T>()          // error
{
    //...
}
```

Çünkü böyle bir bildirimde T tür parametresi, parametre ya da geri dönüş değeri bildiriminde yer almadığı için derleyicinin onun gerçek türünü tespit etmesi mümkün değildir. Halbuki C++ , Java ve C#'ta fonksiyon çağrılırken tür parametresi açıkça belirtilebilmektedir:

```
foo<Int>()             // C++ ve C#'ta bu sentaks var fakat Swift'te yok
```

Generic Sınıflar, Yapılar ve Enum'lar

Nasıl bir fonksiyon ya da metot generic olabiliyorsa bir sınıfın, yapının ya da enum'un tamamı da generic olabilir. Ancak Swift'te protokoller generic olamamaktadır. (Java ve C#'ta arayüzlerin generic olabileceğini anımsayınız). Bir sınıf, yapı ya da enum türünü generic yapabilmek için sınıf, yapı ya da enum isminden sonra açılabilir parantezler içerisinde generic tür parametrelerinin belirtilmesi gerekir. Örneğin:

```
class LinkedList<T> {
    //...
}
```

Bu biçimde bildirilen tür parametreleri tür belirten bir isim olarak sınıf, yapı ve enum bildirimlerinin içerisinde (tabii onların metotlarının da içerisinde) kullanılabilir. Örneğin:

```
class Sample<T>
{
    var a: T

    init(_ a: T)
    {
        self.a = a
    }

    func disp()
    {
        print(a)
    }
}

let s = Sample<Int>(10)

s.disp()
```

Örneğin:

```
import Foundation

struct Stack<T> {
    private var stack: [T]

    init()
    {
        stack = [T]()
    }

    mutating func push(val: T)
    {
        stack.append(val)
    }

    mutating func pop() -> T
    {
        return stack.removeLast()
    }

    var isEmpty: Bool {
        return stack.isEmpty
    }

    var count : Int {
        return stack.count
    }
}
```

```

var s = Stack<Int>()

s.push(val: 10)
s.push(val: 20)
s.push(val: 30)

while !s.isEmpty {
    print(s.pop())
}

```

Generic bir tür kullanılırken tür isminden sonra açılabilir parantezler içerisinde kesinlikle tür argümanlarının belirtilmesi gerekmektedir.

Swift'te (henüz) generic türlerin overload edilmesi özelliği yoktur. Yani aynı isimli biri generic diğeri normal olan iki sınıf, yapı ya da enum birlikte bulunamaz. Benzer biçimde farklı sayıda generic parametresi olan aynı isimli türler de bir arada bulunamazlar. Halbuki C++ buna izin vermektedir. C#'ta da overload işlemi kısmen yapılabilmektedir.

Generic'lerde Tür Kısıtları (Type Constraints)

Generic tür kısıtları Java ve C#'ta da kavram olarak bulunmaktadır. Bir generic fonksiyon ya da sınıf generic tür parametreleri hangi türden açılırsa açılсын anlamlı olmak zorundadır. Aksi takdirde derleme aşamasında error oluşur. Örneğin:

```

func findIndex<T>(<_ a: [T], _ val: T) -> Int?
{
    for i in 0..

```

Burada her T türünün (örneğin her sınıfın ya da yapının) == operatör fonksiyonu olmak zorunda değildir. İşte bu tür durumlarda biz generic parametrelerine bazı kısıtları sağlama zorunluluğu getirebiliriz. Tür parametrelerine kısıt getirme işleminin genel biçimi şöyledir:

<tür parametresi> [: <protokol listesi>]

Genel biçimden de görüldüğü gibi tür parametresini kısıtlama işlemi tür parametresinden sonra ':' atomu ve sonra da protokol listesi getirilerek yapılmaktadır. Örneğin:

```

func findIndex<T: Equatable>(<_ a: [T], _ val: T) -> Int?
{
    for i in 0..

```

```

    }

    return nil
}

let a = [1, 2, 3, 4, 5]
if let result = findIndex(a, 4) {
    print(result)
}
else {
    print("değer bulunamadı")
}

```

Burada derleyiciye T türünün Equatable protokolünü destekleyen bir türle açılacağı garantisi verilmektedir. Artık biz findIndex fonksiyonunu Equatable protokolünü desteklemeyen bir türle çağırmaya çalışırsak derleme aşamasında error oluşur. Örneğin:

```

func findIndex<T: Equatable>(_ a: [T], _ val: T) -> Int?
{
    for i in 0..

```

Equatable protokolü == operatör fonksiyonuna sahiptir. Yani Equatable protokolünü destekleyen bir sınıf ya da yapı == operatör fonksiyonuna sahip olmak zorundadır. Örneğin:

```

func findIndex<T: Equatable>(_ a: [T], _ val: T) -> Int?

```



```

{
    for i in 0..

```

Ancak biz Equatable protokolünü destekleyen bir türü != operatörü ile de kullanabiliriz. Çünkü standart kütüphanede aşağıdaki gibi yazılmış generic bir != operatörü vardır:

```

func !=<T : Equatable>(_ left: T, _ right: T) -> Bool
{
    return !(left == right)
}

```

Başka bir deyişle biz Equatable protokolünü destekleyen bir sınıf, yapı ya da enum türünü == operatörünün yanı sıra != operatörüyle de kullanabiliriz.

Comparable protokolü Equatable protokolünden türetilmiştir. Bu protokolde yalnızca < operatör fonksiyonu vardır. Yani Comparable protokolünü destekleyen bir tür hem == hem de < operatör fonksiyonlarını bulundurmak zorundadır. Bu iki operatör fonksiyonu bulunduğunda kütüphanedeki !=, <=, >= ve > generic operatör metotları devreye girerek != <=, >= ve > işlemlerini == ve < operatörlerini kullanarak yapmaktadır. Dolayısıyla bir tür Comparable protokolünü destekliyorsa biz o türü 6 karşılaştırma operatörüyle de işleme sokabiliriz. Fakat o tür için yalnızca == ve != operatör fonksiyonlarını yazmak zorundayız. Örneğin:

```

func getMax<T: Comparable>(_ a: [T]) -> T
{
    var max = a[0]

    for i in 1..

```

Burada getMax generic fonksiyonunun T tür parametresi Comparable protokolünü desteklemektedir. Dolayısıyla biz bu fonksiyon içerisinde ==, !=, <, >, <= ve >= operatörlerini kullanabiliriz. Swift'in temel türlerinin hepsi zaten Comparable protokolünü desteklemektedir. Örneğin:

```

var a = [23, 45, 28, 54, 98, 12]
var max = getMax(a)
print(max)

```

Örneğin:

```

func getMax<T: Comparable>(_ a: [T]) -> T
{
    var max = a[0]

    for i in 1..

```

```

class Number : Comparable {
    var val: Int

    init(_ val: Int)
    {
        self.val = val
    }

    func disp()
    {
        print(val)
    }
}

```

```

func ==( _ a: Number, _ b: Number) -> Bool
{
    return a.val == b.val
}

```

```

func <(_ a: Number, _ b: Number) -> Bool

```

```

{
    return a.val < b.val
}

var a = [Number(23), Number(45), Number(28), Number(54), Number(98), Number(12)]
var max = getMax(a)
max.disp()

```

Swift'te generic protokol kaavramı yoktur. Ancak generic protokoller dolaylı bir biçimde belli düzeyde oluşturulabilmektedir. Bir prokolde associatedtype anahtar sözcüğü ile bir tür ismi uydurulursa protokolün elemanları ona dayalı olarak oluşturulabilmektedir. Örneğin:

```

protocol P {
    associatedtype T

    func foo(a: T) -> T
}

```

Buradaki associatedtype ile oluşturulmuş olan tür ismine protokolün ilişkin olduğu tür (associated type) denilmektedir. Eğer böyle bir protokolü bir tür destekleyecekse T türü ne olursa olsun yalnızca bir tane protokoldeki kalıba uygun foo metodunu bulundurmak zorundadır. Örneğin:

```

class Sample : P {
    func foo(a: Double) -> Double
    {
        return 0
    }
    //...
}

```

Ya da örneğin:

```

protocol P {
    associatedtype T

    func foo(a: T) -> T
}

class Sample : P {
    func foo(a: Sample) -> Sample
    {
        //...
    }
}

```

Yukarıdaki örnekte artık Sample sınıfı aynı kalıba uygun başka bir foo metodunu içermez. Örneğin:

```

class Sample : P {
    func foo(a: Double) -> Double
    {
        return 0
    }

    func foo(a: Int) -> Int    // error!
    {

```

```

        return 0
    }
}

```

Swift'in standard kütüphanesinde bir grup ExpressibleByXXXLiteral isimli protokol vardır. Bu protokolleri destekleyen türlere XXX kategorisinden sabitler doğrudan atanabilir. Fakat eğer bir sınıf yapı ya da enum bu protokolü destekliyorsa o sınıf yapı ya da enum'un ilgili XXX türünden bir required init metoduna sahip olması gerekmektedir. Örneğin:

```

class Sample : ExpressibleByIntegerLiteral {
    var a: Int
    required init(integerLiteral a: Int)
    {
        self.a = a
    }

    func disp()
    {
        print(self.a)
    }
}

var s: Sample

s = 123    // s = Sample(123)
s.disp()

```

ExpressibleByXXXLiteral protokolleri bir init metodu içermektedir. Bu init metodu XXX kategorisinden sabit değerleri alabilmektedir. Örneğin ExpressibleByIntegerLiteral arayüzü aşağıdaki gibi oluşturulmuştur:

```

protocol ExpressibleByIntegerLiteral {
    typealias IntegerLiteralValue
    init(integerLiteral value: IntegerLiteralType)
}

```

Örneğin:

```

class Sample : ExpressibleByIntegerLiteral {
    var a: Int
    required init(integerLiteral a: Int)
    {
        self.a = a
    }

    func disp()
    {
        print(self.a)
    }
}

var s: Sample

s = 123    // s = Sample(integerLiteral: 123)
s.disp()

```

Biz burada ExpressibleByIntegerLiteral protokolünü destekleyen bir Sample sınıfı yazdık. Böylece artık nokta içermeyen tamsayısal sabitleri Sample türüne doğrudan atayabiliriz. Bu atama işlemi sırasında aslında derleyici Sample sınıfı türünden protokoldeki init metodunu kullanarak bir Sample nesne yaratmaktadır. Yani bu atama işlemi aslında yalnızca kısa bir yazım oluşturmaktadır. C++ ve C#'ta bu işlemin tür dönüştürme operatör fonksiyonlarıyla yapıldığını anımsayınız.

ExpressibleByXXLiteral prorokollerinin listesi şöyledir:

```
ExpressibleByIntegerLiteral
ExpressibleByFloatLiteral
ExpressibleByBooleanLiteral
ExpressibleByNilLiteral
ExpressibleByStringLiteral
ExpressibleByExtendedGraphemeClusterLiteral
ExpressibleByUnicodeScalarLiteral
```

Örneğin sınıf türünden bir referansa string ile değer vermek için ExpressibleByStringLiteral protokolünü desteklememiz gerekir:

```
class Sample : ExpressibleByStringLiteral
{
    var name: String
    required init(stringLiteral name: String)
    {
        self.name = name
    }

    func disp()
    {
        print(name)
    }
}

var s: Sample

s = "Ali"
s.disp()
```