

SWIFT PROGRAMLAMA DİLİ İLE IOS İŞLETİM SİSTEMLERİNDE UYGULAMA GELİŞTİRME

Egitmen: Kaan ASLAN

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 de yapılırken 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 büyütmuşlerdi. 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 işlemcisini çıkartmıştı. Wozniak Altair'den de etkilenerek 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. 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 sürüldü. 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 çatışmaları 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 çıktı. 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 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 Apple'ın Iphone, Ipad ve Ipod cihazlarında kullanılmaktadır. IOS adeta Mac OS X'in mobil versiyonu gibidir. Mac ile IOS ortamlarında 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)

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.

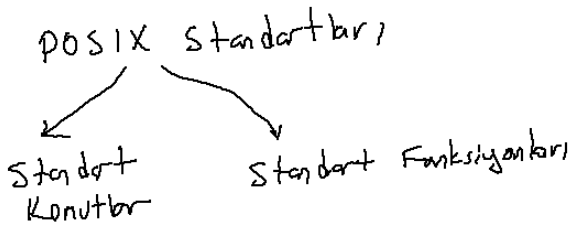
- Mac'lerde Page Up , Page Down, Home ve End tuşları yoktur. Bunların yerine fn ve ok tuşları kullanılır.

- 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.
- Pencereleeri taşımak 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.
- Pencereleerin 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 1.1'inci sürümü çıktı. Bunu da 2015 yılında Swift 2.0 izledi. Kursun verildiği tarih dikkate alındığında Swift'in en son sürümü 2.1'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 gösrerici 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 karşılaşmaktadır.

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

import Swift

print("Merhaba Dünya")
```

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 yorumlayıcısı ve REPL swift komutuyla çalıştırılabilir. swift komutunun yanına dosya ismi verilmezse bir komut satırı karşımıza çıkar. Buna REPL (Read+Evaluate+Print+Run) denilmektedir. REPL denemeler için kullanılabilen karşılıklı etkileşimli bir ortamdır. swift komutunun yanına komut satırı argümanı olarak dosya ismi verilirse o dosyadakiler yorumlama yöntemiyle (yani executable kod üretilmeden) çalıştırmaktadır. Örneğin:

```
swift sample.swift
```

Yorumlama işleminin sonucunda çalıştırılabilir (executable) bir dosya üretilmez. Yorumlayıcı programı okuyarak doğrudan çalıştırmaktadır. Genel olarak Swift programlarının yorumlayıcı yoluyla çalıştırılmasının derleyici yoluyla çalıştırılmasından biraz daha yavaş olduğu söylenebilir. Yorumlama özelliği Swift'tin script tarzı kullanımını kolaylaştırmaktadır.

Merhaba Dünya Programının Açıklaması

Merhaba Dünya programının başındaki import bildirimi Swift isimli modülün programa dahil edileceğini belirtir. import komutu ileride ayrıntılarıyla ele alınacaktır. Bu sayede biz bu modüldeki isimleri niteliklendirmeden doğrudan kullanabiliriz. print Swift'in standart kütüphanesine ilişkin bir fonksiyondur. Swift yukarıda da sözü edildiği gibi çok modelli (multi paradigm) bir programlama dilidir. Dolayısıyla biz onu prosedürel biçimde (yani sınıf olmadan fonksiyonlarla) kullanabiliriz.

Swift programları nereden çalışmaya başlar? Pek çok dilde main isminde bir başlangıç fonksiyonu (entry point function) bulunmaktadır. Swift'te program kaynak kodun tepesinden çalışmaya başlar. (Bu konuda bazı ayrıntılar var. İleride ele alınacaktır.) Dolayısıyla Merhaba Dünya programında akış kaynak dosyanın tepesinden başlar. Orada print fonksiyonu çağırılmıştır.

Atom (Token) Kavramı

Bir programlama dilindeki eklenti başına anlamlı olan en küçük birime atom (token) denilmektedir. Program aslında atomların bir araya getirilmesiyle oluşturulur. Atomlar doğal dillerdeki sözcüklere benzetilebilirler. Derleyiciler de derleme işleminin ilk aşamasında kaynak kodu atomlarına ayırmaktadır.

Atomlar 6 gruba ayrılırlar:

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) Ayraçlar (delimiters / punctuators): Yukarıdaki atom gruplarının dışında kalan ';' gibi , '{' gibi atomlara ayraç 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 bulunuyorsa (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
}
else {
    --total
} print(total) // geçersiz!
```

Fakat:

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

Swift'in Temel Veri Türleri

Tür bir değişkenin bellekte kaç byte yer kapladığını ve ona hangi aralıkta ve formatta değer yerleştirilebileceğini belirten önemli bir bilgidir. Swift 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 işaretsiz biçimi UInt32 biçimindedir.
- Float türü IEEE 754 "short real format"a uygundur. Yuvarlama hatalarına direnci bu türün 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. C, C++, Java ve C#'taki double türüyle aynı formata sahiptir.
- Character türü tek bir karakteri tutmak için düşünülmüştür.
- 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 kütüphane sınıfıdır.

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 arınmaktadır.

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.

Bu ana versiyon numaralarının dışında Apple küçük numaralı sürümler de çıkarmıştır. Şu anda XCode'un en son sürümü 7.2 sürümüdür.

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 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 İle 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
```

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

Örneğin:

```
var a = 100, b = 200
```

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

Örneğin:

```
var a: Int = 100, b: Double = 200
```

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

Örneğin:

```
var a = 100, b: Double = 10.2;
```

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

Örneğin:

```
var a: Int, b: Double;
```

```
a = 10;
b = 20.3;
```

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

Görüldüğü var ile bildirim yapılırken değişken isminden sonra tür belirtilmiyorsa ilkdeğer verme zorunludur. Çünkü değişkenin türü bu durumda ancak ilkdeğer ifadesinden çıkarsanabilmektedir. Fakat değişken isminden sonra tür belirtilirse bu durumda ilkdeğer verme zorunlu olmaz. let ile yapılan bildirimlerde tür belirtilse bile ilkdeğer verme zorunludur. Örneğin:

```
let x: Int;           // error
x = 10;
```

(Bu son örneğin swift derleyicileri tarafından kabul edildiğini görürseniz şaşırmayın. Ancak Swift'in resmi dokümanlarında geçersiz olduğu belirtilmiştir.)

Eğer bildirimde değişkenin türü belirtilmemişse verilen ilkdeğerden tür tespiti yapılır. Verilen ilkdeğer nokta içeriyorsa tür Double olarak, içermiyorsa Int olarak belirlenmektedir. Örneğin:

```
var a = 100, b = 12.2;
```

Burada a Int türden b ise Double türündendir. İki tırnakla ilkdeğer verme durumunda tür String olarak belirlenir. Örneğin:

```
var s = "ankara";      // s String türünden
```

Swift'te Character türü de iki tırnakla (tabi iki tırnak içerisinde tek bir karakter bulunmalı) ifade edilmektedir. Ancak ikş tırnak ifadeleri otomatik olarak String türü olarak tespit edilirler. Örneğin:

```
var x = "a"            // x Character türünden değil String türünden
```

Tabi bildirim sırasında değişkenin türü belirtilerek iki tırnak ifadesinin Character biçiminde ele alınması sağlanabilir. Örneğin:

```
var ch: Character = "a"      // ch Character türünden
```

Ya da aynı işlem şöyle de yapılabilirdi:

```
var ch = Character("a")      // ch Character türünden
```

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

```
var a = 33 == 34;           // a Bool türden
```

Değişken let ile bildirilirse const hale gelir. Dolayısıyla bu değişkene bir daha değer ataamayız. Örneğin:

```
let a = 100
a = 200                     // error!
```

Anahtar Notlar: print fonksiyonunda parametrik yazım için \ (değişken ismi) kalıbı kullanılmaktadır. Önce bir \ karakteri sonra da parantezler içerisinde değişken ismi belirtilirse bu o değişkenin değeri anlamına gelir. Örneğin:

```
var a = 10

print("a = \(a)")
```

Örneğin:

```
var a: Int = 10, b: Double = 10.5
```

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

Fonksiyon Bildirimleri

Swift'te fonksiyonlar C# ve Java'da olduğu gibi bir sınıfın içerisinde bulunmak zorunda değildir. Bu bakımdan Swift C++'a benzetilmiştir. 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 add(a: Int, b: Int) -> Int  
{  
    return a + b  
}
```

```
var x = add(10, b: 20)
```

```
print(x)
```

Örneğin:

```
func foo()  
{  
    print("foo")  
}
```

```
foo()
```

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 tuple) 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: Double, b: Double, c: Double) -> Double  
{  
    return a + b + c  
}
```

```
}
```

```
var result = add(100, b: 200, c: 300)
print(result)
```

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

Geri dönüş değeri olmayan bir fonksiyonda biz return kullanmak zorunda değiliz. Zaten akış fonksiyonun sonuna geldiğinde fonksiyon sonlanacaktır. Ancak geri dönüş değeri mevcut olan bir fonksiyonda return kullanılmak zorundadır. Fonksiyonun akışı onun her mümkün akışı için return deyimini görmesi gerekir.

Örneğin:

Fonksiyon parametre bildiriminde parametrenin türleri kesinlikle belirtilmek zorundadır. Örneğin:

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

Parametre değişken bildiriminde var ya da let anahtar sözcükleri kullanılabilir. Bunların kullanılmadığı durumda default olarak sanki let kullanılmış gibi işlem görür. var anahtar sözcüğü parametre değişkeninin fonksiyon içerisinde değiştirilebileceğini, let ise değiştirilemeyeceğini belirtir. Default olarak parametre değişkenleri let durumdadır yani fonksiyon içerisinde onların değerleri değiştirilemez. (Örneğin C, C++, Java ve C#'ta default olarak biz parametre değişkenlerinin değerlerini istersek değiştirebiliriz.)

Fonksiyonların Çağırılması

Bir fonksiyon çağrılırken parametre değişkeni sayısı kadar argüman girilmek zorundadır. Argümanlar aynı türden herhangi bir ifade olabilirler. Parametrelerin birer yerel ismi (local name) ve birer de dışsal ismi (external name) bulunur. Fonksiyon çağrılırken argümanda etiket (label) olarak dışsal isim belirtilmek zorundadır. Dışsal isimler fonksiyon parametre bildiriminde yerel isimlerin solunda belirtilirler. Örneğin:

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

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

Görüldüğü gibi fonksiyon çağrılırken dışsal isimler etiket olarak belirtilmek zorundadır. Fonksiyon çağırma sırasında argüman oluşturma işleminin genel biçimi şöyledir:

[dışsal isim etiketi][:]<ifade>

Dışsal isim yerine '_' karakteri kullanılırsa bu durumda çağırma sırasında dışsal isim etiketi bulundurulmaz. Örneğin:

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

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

Fonksiyonların birinci parametrelerinde default olarak '_' belirlemesi yapılmış gibidir. Yani biz birinci parametreye dışsal isim vermezsek bunun için argümanda etiketleme yapmamalıyız. Örneğin:

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

Bu bildirim şununla eşdeğerdir:

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

Fonksiyonun çağırımı şöyle yapılabilir:

```
var result = foo(10, height: 200) // birinci argümanda etiketleme yapamayız
print(result)
```

Eğer fonksiyonun birinci parametresine dışsal isim verilmişse artık çağırırken bunun etiketlendirilmesi gerekir. Örneğin:

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

var result = foo(10, height: 200) // error! birinci argümanda etiketleme yapmak zorundayız
print(result)
```

Anahtar Notlar: İstisna olarak Sınıfların başlangıç metotlarının (init metotlarının) birinci parametresi '_' biçiminde değildir. init metotlarını çağırırken ilk argüman da etiketlendirmek zorundadır. Sınıflar ileride ayrıntılarıyla ele alınacaktır.

Fonksiyon bildirim sırasında parametre değişkenlerine dışsal isim hiç verilmeyebilir. Bu durumda yerel isim aynı zamanda dışsal isim olarak kullanılır. Örneğin:

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

var result = foo(10, h: 200)
print(result)
```

Burada yine birinci parametre için dışsal isim olarak '_' belirleyicisi kullanılmıştır. Ancak ikinci parametrede dışsal isim kullanılmadığı için yerel isim aynı zamanda dışsal isim olarak ele alınmaktadır.

Swift'te Fonksiyon Overload İşlemleri

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

Swift'te aynı isimli fonksiyonlar bulunabilir. Ancak onların arasında aşağıdaki farklılıkların biri ya da birden fazlasının olması gerekir:

- Parametre sayılarının farklı olması
- Parametre türlerinin farklılığı
- Dışsal isimlerin farklılığı
- Geri dönüş değerinin farklılığı

Örneğin:

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

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

Burada yukarıda belirtilen farklılıkların hiçbiri söz konusu değildir. İki foo fonksiyonunun da dışsal isimleri aynıdır (her ikisinde _ biçimindedir). Örneğin:

```
import Swift

func foo(x a: Int)
{
    print("foo 1")
}

func foo(y b: Int)
{
    print("foo 2")
}

var result: Int
foo(x: 10)      // foo 1
foo(y: 20)      // foo 2
```

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

```
func foo(x a: Int)
{
```



```

    print("foo 1")
}

func foo(x b: Double)
{
    print("foo 2")
}

var result: Int
foo(x: 10)           // foo 1
foo(x: 20.4)         // foo 2

```

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

```

func foo(x a: Int)->Int
{
    print("foo 1")

    return 100
}

func foo(x b: Int)->Double
{
    print("foo 2")

    return 100.0
}

var x: Int, y: Double

x = foo(x: 0)        // foo 1
print(x)

y = foo(x: 0)        // foo 2
print(y)

```

Yukarıdaki iki foo'nun da dışsal 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. Tabi bunu her zaman başaramayabilir. Örneğin:

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

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

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

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

Swift'te de çağrılan fonksiyonun geri dönüş değeri kullanılmak zorunda değildir. Yani fonksiyonun geri dönüş değerinden biz faydalanabiliriz ya da hiç faydalanamayabiliriz.

Fonksiyon Parametrelerinin Default Değer Alması Durumu

Default argüman C++'ta öteden beri vardı. C#'a Microsoft Language Specification 3.0 versiyonuyla sokulmuştur. Bu özellik Swift'te de aktarılmıştır.

Fonksiyon bildiriminde parametre değişkenine 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(10, b: 20)
foo(10)          // foo(10, b: 100)
```

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

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

foo(10, b: 20) // foo(10, b: 20)
foo(10)        // foo(10, b: 200)
foo()          // foo(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 argümanlı fonksiyonlar Swift'in standart kütüphanesinde de karşımıza çıkmaktadır. Örneğin aslında print fonksiyonu da terminator dışsal isimli bir parametreye sahiptir. Bu parametrenin default değeri "\n" biçimdedir. Bu parametre yazdırma işleminden sonra ekstra olarak hangi karakterlerin ekrana basılacağını belirtir. Başka bir deyişle:

```
print(a)
```

ile

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

eşdeğer işleme yol açar. Örneğin:

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

Default argümana sahip fonksiyonla aynı isimli daha az parametreye sahip fonksiyonlar overload edilebilir. Fakat bunların çağrılması sırasında iki anlamlılık (ambiguity) 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 dışsal isimler 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 değişkenşne default değer vermek için o değer çok yaygın kullanılıyor olması gerekir. Aksi taktirde kodu inceleyen kişiler yanlış izlenimlere kapılabilirler.

İç İçe Fonksiyon Bildirimleri

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

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

İçerde bildirilmiş fonksiyon ancak içinde bildirildiği fonksiyon tarafından çağrılabilir. Yani yukarıdaki örnekte bar fonksiyonu yalnızca foo fonksiyonu tarafından çağrılır.

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()
{
    var x = 10

    func bar()
    {
        x = 20
    }

    print(x)
}

foo()          // 10
```

Bu örnekte bar fonksiyonu foo fonksiyonunun içerisinde bildirilmiştir fakat çağrılmamıştır. Tabi foo fonksiyonu bar'ı çağıracaksa ancak bildiriminden sonra çağırabilir. Örneğin:

```
func foo()
{
    var x = 10

    func bar()
    {
        x = 20
    }

    bar()

    print(x)
}

foo()          // 20
```

Sabitler

Swift'te C, C++, Java ve C'taki gibi sabitlerin türleri yoktur. Sabitler değişkenlere atanırken onlar için uygunluk değerlendirmesi yapılmaktadır. 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:

```
var x = 0b1010101          // 2lik sistemde
```

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

```
var x = 0o12345             // 8'lik sistemde
```

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

Sayının kaçlık sistemde yazıldığıнын tür bakımından bir önemi yoktur.

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

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 Character türünden sabitler de iki tırnak ile belirtilmektedir. Pekiyi iki tırnak içerisine biz tek bir karakter yazarsak bu String sabit olarak mı yoksa Character sabiti olarak mı değerlendirilecektir? İşte eğer biz var ya da let bildiriminde hiç tür belirtmemişsek default olarak Swift'te iki tırnak içerisinde bulunan tek karakter String olarak değerlendirilir. Örneğin:

```
var a = "a"              // a String türünden
```

Eğer biz türü Character olarak belirlemişsek bu durumda iki tırnak içerisindeki tek karakter Character sabiti olarak değerlendirilir. Örneğin:

```
var ch: Character = "a"   // geçerli
```

Tabi iki tırnak içerisinde birden fazla karakter varsa zaten o sabitin Character olma olasılığı yoktur. Örneğin:

```
var ch: Character = "ab"  // error!
```

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

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!
```

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:

```
var result = foo()  
if result != nil {  
    //...  
}  
else {  
    //...  
}
```

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

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

a = 10 // geçerli
b = a // error
b = a! // geçerli
```

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

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

a = nil // geçerli
b = a! // geçerli, fakat exception oluşacak
```

O halde biz T? türündeki değişkenin içerisinde nil olmadığından emin olarak onu "unwrap" yapmalıyız. Örneğin:

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

a = nil // geçerli

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

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" siminde 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'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 açar. 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 işlem bir tür dönüştürmesi (type cast) fakat şimdilik öyleymiş gibi düşünülebilirsiniz. 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)
```

Fakat böylesi bir dönüştürme sırasında kaynak değer hedef türün sınırları içerisinde kalmıyorsa çalışma zamanı sırasında exception oluşur. Örneğin:

```
var a: Int8, b: Int32 = 1000

a = Int8(b)    // program başarılı derlenir, fakat exception oluşur
print(a)
```

Swift'te farklı türleri de işleme sokamayız. Örneğin C# ve Java'da biz int bir değerle double bir değeri toplayabiliriz. Bu dillerde "işlem öncesi otomatik tür dönüştürmesiyle" küçük tür büyük türe dönüştürülür ve sonuç büyük tür türüne 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
```

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.

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

2) Tek operandlı operatörlerle 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
d = a ++     // error! tek operandlı operatörler operandlarıyla bitişik yazılmak zorundadır
d = a++* b   // error ++'nın sağında boşluk karakteri yok!
a = !!b      // error!
a = !(!b)    // geçerli
```

Swift'te Operatör Öncelikleri

Bilindiği gibi operatörler paralel işleme sokulmaz. 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 yacak makina komutlarını bu sırada oluşturmaktadır. Swift'teki operatörlerin çoğu zaten diğer yaygın dillerdekilerle aynıdır. Fakat operatör öncelikleri bakımından Swift ile diğerleri 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!
```

Swift'te operatörler konusundaki bazı ayrıntılar şunlardır:

- Karşılaştırma operatörleri Bool türden sonuç üretmektedir.
- Mantıksal operatörlerin operandı Bool türden olmak zorundadır.
- Mantıksal && ve || operatörlerinin yine Swift'te de kısa devre (short circuit) özelliği vardır. Bunların sol tarafı önce yapılır. Duruma göre sağ tarafları hiç yapılmayabilir.
- Bilindiği gibi bir işlem sonucunda elde edilen değerin sonucun ilişkin olduğu türün sınırları içerisinde kalmaması durumuna "taşma (overflow)" denilmektedir. Taşma yukarıdan ya da aşağıdan olabilir. (Aşağıdan taşmalara "underflow" denilse de yalnızca "overflow" sözcüğü de her ikisini kapsar biçimde kullanılmaktadır.) Aritmetik operatörlerde taşma durumunda exception oluşmaktadır. İşte taşma durumunda exception oluşmadan sayının yüksek anlamlı byte'larının atılarak işleme devam edilmesi isteniyorsa +, -, *, / ve % operatörlerinin &+, &-, &*, &/ ve &% biçimindeki versiyonları kullanılmalıdır. Örneğin:

```
var a: Int8 = 100, b: Int8 = 100, c: Int8
```

```
c = a &+ b
print(c)           // -56
```

Yukarıdaki örnekte biz &+ yerine normal + operatörünü kullansaydık taşma nedeniyle exception oluşup programımız çökecekti.

- Swift'te ... (ellipsis) operatörüne kapalı aralık (closed interval), ..< operatörüne ise yarı açık aralık operatörü denilmektedir. Bu operatörler iki operandlı aralık operatörlerdir. Bunların operandları tamsayı türlerine ilişkin olabilir. (Aslında ForwardIndexType isimli protokolü destekleyen herhangi bir tür türünden olabilir.) a ... b işlemi ya da a ..< işlemi aslında Range<T> türünden bir nesnenin oluşturulmasına yol açar. Yani örneğin:

```
var x = 1...10
```

bildirim ile:

```
var y: Range<Int> = Range(start: 1, end: 11)
```

eşdeğerdir. a...b ifadesi a'dan b'ye (b dahil olmak üzere) değerlerden oluşan bir collection nesne gibi düşünülmelidir. a..<b ifadesi ise a'dan b'ye (b dahil değil) değerlerden oluşan bir collection nesne gibi düşünülebilir.

..
< operatöründen sağ taraftaki değer aralığa dail değildir. Örneğin:

```
var x = 1...10

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

Aralık nesneleri başta for-in döngüsü olmak üzere pek çok yerde kullanılmaktadır. Örneğin switch deyiminde de case bölümlerinde aralıklar kullanılabilir.

?? operatörüne İngilizce "nil coalescing operator" denilmektedir. Bu operatör 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 operatörünün (?:) kullanımı diğer dillerle aynı biçimdedir.

- ++ ve -- operatörlerinin kullanımı da diğer dillerde olduğu gibidir. Yani bunlar tek operandlı hem önek hem de sonek olarak kullanılabilen operatörlerdir. Sonek kullanımda sonraki işleme artırılmamış ya da eksiltilmemiş değer girer. Örneğin:

```
var a, b: Int

a = 3
b = ++a * 2
print("a = \({a}), b = \({b})")    // a = 4, b = 8

a = 3
b = a++ * 2
print("a = \({a}), b = \({b})")    // a = 4, b = 6
```

- % operatörü C# ve Java'daki gibidir. Bu operatör sol taraftaki operandın sağ taraftaki operanda bölümünden elde edilen kalan değerini üretir. Operand'lar gerçek sayı türlerine ilişkin olabilir. Negatif değerın pozitif değere bölümünden elde edilen kalan negatiftir.

- Swift'te == ve ===, != ve !== operatörleri arasında küçük bir farklılık vardır. == operatörü overload edilebilir. Bu durumda iki nesnenin içinde aynı değerler var mı diye bakar. Halbuki === operatörü operandı olan referansların aynı nesneyi gösterip göstermediğine bakmaktadır. (Dolayısıyla yapılar zaten farklı nesnelerdir. İki yapı değişkeni için === operatörü zaten false verir). == ve === operatörlerinin kullanımı ileride örneklerle ele alınacaktır.

Swift'te Klavyeden Okuma Yapmak

Swift'te klavyeden (stdin dosyasından) okuma yapmak için birkaç yöntem kullanılabilir. Fakat nihayet Swift'in standart kütüphanesine readLine global fonksiyonu da yerleştirilmiştir. Bu sayede okuma yapmak çok daha kolaylaşmıştır. Birinci yöntem Cocoa sınıflarından faydalanmaktır. Örneğin:

```
var keyboard = NSFileHandle.fileHandleWithStandardInput()
var inputData = keyboard.availableData
NSString str = NSString(data: inputData, encoding:NSUTF8StringEncoding)
```

İkinci yöntem Objective-C'de okuma yapan bir fonksiyon yazıp bunu Swift'ten çağırmak olabilir. Aslında birinci yöntemde de dolaylı olarak böyle bir teknik uygulanmaktadır. Objective-C ile (ya da C ile) Swift arasında bir karşılıklı kullanım (interoperability) vardır. Swift ile Objective-C arasındaki karşılıklı kullanım hakkında ileride temel bilgiler verilecektir. Ancak böyle bir okuma sırasıyla şu aşamalardan geçilerek yapılabilir:

1) Projeye bir tane .m uzantılı Objective-C kaynak dosyası eklenir. Ekleme sırasında IDE bize köprü başlık dosyasınının (bridging header file) da oluşturulup oluşturulmayacağını soracaktır. Bizim oluşturulmasını istememiz gerekir.

2. Oluşturulan .m dosyasına aşağıdaki C kodu yazılır:

```
#include <stdio.h>

int getInt()
{
    int val;

    scanf("%d", &val);

    return val;
}
```

3. Köprü başlık dosyasına C fonksiyonun prototipi eklenir:

```
int getInt(void);
```

4. Swift'ten fonksiyon doğrudan kullanılır.

Neyse ki Swift'e readLine fonksiyonu eklendiği için artık okuma doğrudan Swift içerisinde de yapılabilmektedir. readLine fonksiyonu stdin dosyasından '\n' karakterini görene kadar okuma yapar ve okunan yazıyı bize String? olarak verir. (Çünkü stdin bir dosyaya yönlendirilmiş olabilir ve dosya sonuna gelinmiş olabilir. Bu durumda readLine başarısız olabilecektir. İşte bu nedenle fonksiyonun geri dönüş değeri String değil de String? biçimindedir.) Örneğin:

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



```
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()!!)

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: Double = b * b - 4 * a * c

    if delta < 0 {
        print("kök yok!")
    }
    else {
        var x1, x2: Double

        x1 = (-b + sqrt(delta)) / (2 * a)
        x2 = (-b - sqrt(delta)) / (2 * a)

        print("x1 = \(x1), x2 = \(x2)")
    }
}

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)
```

Maalesef Swift'in standart kütüphanesinde temel matematiksel işlemleri yapan fonksiyonlar henüz yoktur. (Muhtemelen unutulmuştur) Bu yüzden örneğimizde Cocoa'dan gelen sqrt fonksiyonu kullandık.

if deyiminin seçeneksel türlerle kullanılan değişik bir biçimi de vardır. Bu biçimde if anahtar sözcüğünü bir bildirim izler. Bu bildirimde değişkene seçeneksel bir türden ilkdeğer vermek zorunludur. Eğer bu seçeneksel türe verilen ilkdeğer nil değilse if doğrudan sapar, nil ise yanlıştan sapar. Örneğin:

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

    return sqrt(a)
}

var val = -23.0

if let result = mysqrt(val) {
    print("\(result)")
}
else {
    print("negatif sayıların gerçek kökü olmaz!")
}
```

Tabi burada bildirim let yerine var anahtar sözcüğüyle yapılabilirdi. Eğer bildirim var ile yapılırsa biz bildirilen değişkene if deyiminin doğruysa kısmında değer atayabiliriz. Her hülükarda burada bildirilen değişkenin faaliyet alanı if'in doğruysa kısmına ilişkindir.

Bu biçimdeki if deyiminde verilen ilkdeğerin seçeneksel türe ilişkin olması zorunludur. Ayrıca bildirimdeki türün artık seçeneksel olmadığını da vurgulayalım. Yani örneğimizdeki result Double türündendir, Double? türünden değildir. Tabi istenirse bildirim sırasında tür de açıkça belirtilebilir:

```
if let result: Double = mysqrt(val) {
    print("\(result)")
}
else {
    print("negatif sayıların gerçek kökü olmaz!")
}
```

Yukarıdaki ifadenin eşdeğeri şöyle oluşturulabilir:

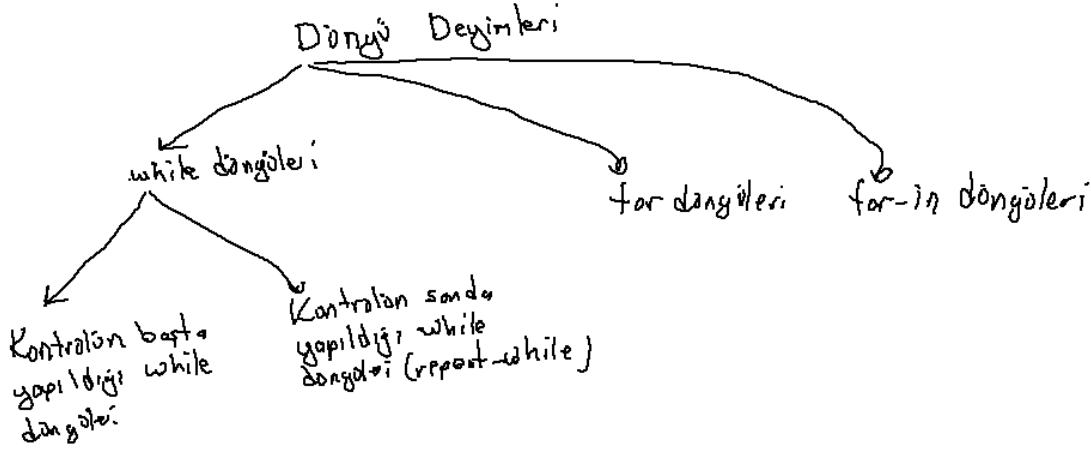
```
var val = -23.0
var result: Double?

result = mysqrt(val)

if result != nil {
    print("\(result!)")
}
else {
    print("negatif sayıların gerçek kökü olmaz!")
}
```

Döngü Deyimleri

Swift'te üç döngü deyimi vardır: while döngüleri, for döngüleri ve for-in döngüleri. Yine while döngüleri kendi aralarında "kontrolün başta yapıldığı while döngüleri" ve "kontrolün sonda yapıldığı while döngüleri" biçiminde ikiye ayrılmaktadır.



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: Int = 0  
  
while i < 10 {  
    print("\(i) ", terminator:"")  
    ++i  
}  
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'da do-while yerine repeat-while anahtar sözcükleri tercih edilmiştir. Semantik bir değişiklik yoktur. Döngünün genel biçimi şöyledir:

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


Örneğin:

```
var i = 0

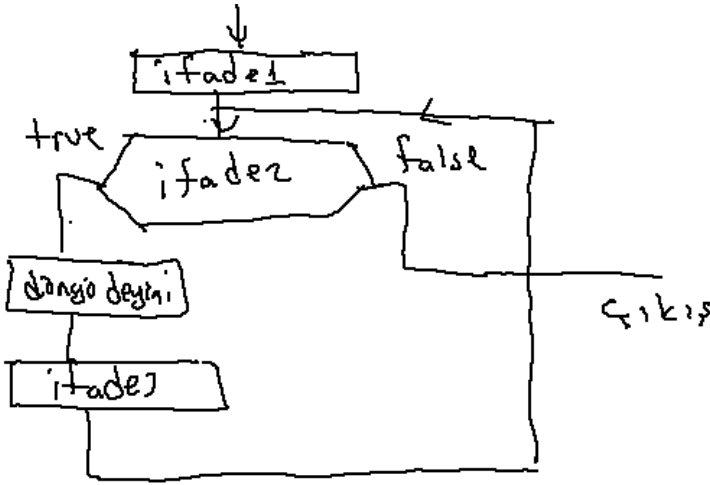
repeat {
  print(i)
  ++i
} while i < 10
```

for Döngüleri

for döngüleri diğer dillerdeki for döngülerine çok benzemektedir. Genel biçimleri şöyledir:

```
for [( ) [ifade1]; [ifade2]; [ifade3] ( )] {
  //...
}
```

Döngünün çalışması aşağıdaki akış diyagramında belirtildiği gibidir:



Döngünün birinci kısmındaki ifade döngüye girişte yalnızca bir kez yapılır. Döngü ikinci kısımdaki ifade true olduğu sürece yinelenir. Her döngü deyimi çalıştırıldığında üçüncü kısımdaki ifade bir kez yapılır. Örneğin:

```
var i: Int

for i = 0; i < 10; ++i {
  print(i)
}
```

Döngünün ikinci kısmındaki ifade Bool türden olmak zorundadır. C++, Java ve C#'ta olduğu gibi birinci kısımda bildirim yapılabilir.

```
for var i = 0; i < 10; ++i {
  print(i)
}
```

Birinci kısımda bildirilen değişkenler yalnızca for döngüsünün içerisinde faaliyet gösterirler.

for döngüsünün birinci ve üçüncü kısmı virgül atomu ile genişletilebilir. Örneğin:

```
for var i = 0, k = 100; i + k > 50; ++i, k -= 2 {  
    print(i, separator: "")  
}  
print("")
```

Örneğin:

```
func addTotal(lastVal: Int) -> Int  
{  
    var i, total: Int  
  
    for i = 1, total = 0; i <= lastVal; total += i, ++i {  
    }  
  
    return total  
}  
  
var result = addTotal(100)  
print(result)
```

for döngüsünün herhangi bir ya da birden fazla kısmı bulunmak zorunda değildir. Örneğin:

```
for var i = 0; i < 100; ++i {  
    print(i)  
}
```

ile aşağıdaki döngü:

```
var i = 0  
for ; i < 100; ++i {  
    print(i)  
}
```

ve aşağıdaki döngü:

```
for var i = 0; i < 100; {  
    print(i)  
    ++i  
}
```

eşdeğerdir.

Birinci ve üçüncü kısmı olmayan for döngüleri while döngüleriyle eşdeğerdir. Örneğin:

```
for ; ifade ; {  
    //...  
}
```

ile,

```
while ifade {
```

```
    //...
}
```

tamamen eşdeğerdir. Görüldüğü gibi for döngüleri aslında while döngülerinin daha genel bir biçimidir. Aşağıdaki gibi bir for döngüsü olsun:

```
for ifade1; ifade2; ifade3 {
    //...
}
```

bunun while eşdeğeri şöyle yazılabilir:

```
ifade1
while ifade2 {
    //...
    ifade3
}
```

for döngüsünün ikinci kısmı bulundurulmazsa koşulun her zaman sağlandığı kabul edilir. Örneğin:

```
for var i = 0;; ++i {          // dikkat sonsuz döngü!
    print(i)
}
```

Yani döngünün ikinci kısmına birşey yazmamakla oraya true yazmak aynı anlama gelir. Örneğin aşağıdaki işlem yukarıdakiyle eşdeğerdir:

```
for var i = 0; true; ++i {      // dikkat sonsuz döngü!
    print(i)
}
```

for döngüsünün hiçbir kısmı bulunmayabilir. Böylece saf bir sonsuz döngü şöyle ifade edilebilir:

```
for ;; {          // sonsuz döngü
    //...
}
```

Bu biçimde sonsuz döngü while ile de oluşturulabilirdi:

```
while true {
    //...
}
```

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

```

    if val == 0 {
        break
    }
    print(val * val)
}

```

Burada programın akışı break deyimini gördüğünde döngü sonlanır. Örneğin:

```

import Foundation

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

```

Burada programın akışı continue anahtar sözcüğünü gördüğünde döngü yeni bir yinelemeye girer.

İç içe döngülerde break ve continue deyimleri yalnızca kendi döngüsü için işlem yapar. Ancak Swift'te Java'da olduğu gibi etiketli break ve continue deyimleri de vardır. Böylece iç içe döngü de 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 var i = 0; i < 10; ++i {
    for var k = 0; k < 10; ++k {
        print("(\\(i), \\(k))", terminator: "")
        str = readLine()!
        if str == "q" {
            break all
        }
    }
}

```

Aynı örnek continue için de verilebilir:

```

var str: String

all:
for var i = 0; i < 10; ++i {
    for var k = 0; k < 10; ++k {
        print("(\\(i), \\(k))", terminator: "")
        str = readLine()!
        if str == "q" {
            continue all
        }
    }
}

```

```

    }
}
}

```

switch Deyimi

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

```

switch ifade {
    case <ifade listesi>:
        //...
    case <ifade listesi>:
        //...
    case <ifade listesi>:
        //...
    default:
        //...
}

```

switch deyimi case bölümlerinden ve default bölümünden oluşur. Deyim şöyle çalışmaktadır: Derleyici switch anahtar sözcüğünün yanındaki ifadenin değerini hesaplar. Sonra yukarıdan aşağıya doğru case bölümlerini inceler. Akışı ilk uygun olan case bölümüne aktarır. Eğer uygun case bölümü bulunamazsa default bölüm çalıştırılır. Örneğin:

```

var val: Int
print("Bir sayı giriniz:", terminator:"")
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.) Default durumda switch deyiminde "fall through" yoktur. Ancak case bölümlerinde fallthrough anahtar sözcüğü kullanılırsa akış aşağıya doğru düşer. Örneğin:

```

var val: Int
print("Bir sayı giriniz:", terminator:"")
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          // warning
    ifade2
case 2:
    //...

```

Tabi bir koşul altında da fallthrough uygulanabilir. Örneğin:

```

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

```

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 kullanıma 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'in 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 x: Int, y: Int, val: Int
```

```
print("x:", terminator:"")
```

```
x = Int(readLine()!!)
```

```
print("y:", terminator:"")
```

```
y = Int(readLine()!!)
```

```
print("val:", terminator:"")
```

```
val = Int(readLine()!!)
```

```
switch val {
```

```
    case x:
```

```
        print("birinci case")
```

```
    case y:
```

```
        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 seçilmiştir")
}
```

Swift'in switch deyiminde case bölümleri where koşul cümlesi ile oluşturulabilir. Örneğin:

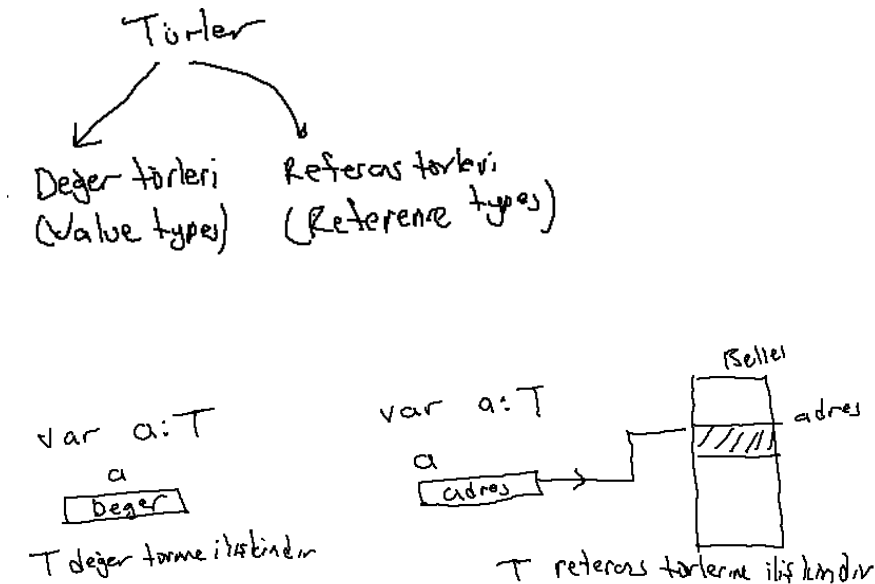
```
var val: Double
print("Bir sayı giriniz:", terminator:"")
val = Double(readLine()!)

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")
}
```

Değer Türleri ve Referans Türleri

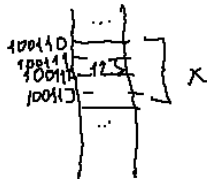
Tıpkı C#'ta olduğu gibi Swift'te de türler kategori olarak iki kısma ayrılmaktadır: Değer türleri (value types) ve referans türleri (reference types). T bir tür olmak üzere T türünden bir değişken eğer değerin doğrudan kendisini

tutuyorsa T türü kategori olarak değer türlerine ilişkindir. Eğer T türünden değişken değeri değil de değerin bulunduğu bellek adresini tutuyorsa T 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: Int = 123
```



Burada x'in adresi 100110'dur.

Java, C# ve Swift'teki referans türleri aslında C ce 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.

Swift'te Diziler

Elemanları aynı türden olan ve bellekte ardışıl biçimde bulunan veri yapılarına dizi (array) denir. Diziye 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 gösterilir. 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 iki biçimde yaratılır:

1) Array<T> yapısının başlangıç metodu yoluyla: 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>()
```

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

```
var a:[Int] = [Int](count: 10, repeatedValue: 100)
```

Burada a dizisi 10 elemanlı olarak yaratılacaktır ve dizinin her elemanında 100 bulunacaktır.

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

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

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)
```

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

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

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

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 verilen ilkdeğerlerden biri Double olarak değerlendirildiği için dizi de Double türünden kabul edilir. Köşeli parantezler içerisinde noktasız tamsayı ya da noktalı gerçek sayı dışında başka türler varsa dizi NSObject türünden olur. Örneğin:

```
var a = [1, "Ali", 3] // a değişkeni [NSObject] türünden
print(a.dynamicType)
```

Aslında bu durumda dizinin NSObject türünden olması Swift referans kitabında belirtilmemektedir. Ayrıca böylesi köşeli parantezlerin [NSObject] olarak yorumlanabilmesi için Foundation modülü de import edilmelidir.

Köşeli parantez yoluyla boş bir dizi de yaratılabilir. Örneğin:

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

Dizi elemanlarına diğer dillerin çoğunda olduğu gibi [] operatörü ile erişilmektedir. Dizinin ilk elemanı sıfırıncı indekslidir. 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 gibidir. 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:[Int] = [1, 2, 3, 4, 5]
a[0...1] = [10, 20]
```

```
print(a)    // [10, 20, 3, 4, 5]
```

Örneğin:

```
var a:[Int] = [1, 2, 3, 4, 5]
var b = a[0...2]           // b ArraySlice<Int> tütünden
print(b)                   // [1, 2, 3]
```

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

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
```

Dizilerin uzunlukları da tıpkı Java ve C#'ta olduğu gibi dizi nesnesinin içerisinde saklanmaktadır. Dizi uzunluğu Array<T> yapısının count propert'si yoluyla elde edilmektedir. Örneğin:

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

Örneğin:

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

    for var i = 1; i < a.count; ++i {
        if a[i] > max {
            max = a[i]
        }
    }

    return max
}

var a:[Int]

a = [120, 23, 17, 21, 9, -5, 34]
print(getMax(a))
```

Swift'in dizileri zaten büyütülebilen bir yapıdır. Büyütme işlemi kapasite artırımı ile yapılır. Kapasite dizi elemanları için tahsis edilmiş olan uzunluktur. Count ise dizideki dolu olan eleman sayısıdır. Diziye eleman eklendiğinde Count bir artar. Count değeri Kapasite değerine geldiğinde tahsis edilmiş alan artırılmaktadır. Yani Array<T> yapısı kendis 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.

Dizi için ayrılan alan `Array<T>` yapısının `capacity` property'si ile elde edilebilir. `capacity` property'si C#'ın aksine read-only'dir. Yani bunun değerine değiştiremeyiz. Ancak yapının `reserveCapacity` metodu bu işi yapmaktadır. Örneğin:

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

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

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

```
mutating func append(_ newElement: Element)
```

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

```
mutating func insert(_ newElement: Element, atIndex i: Int)
```

Örneğin:

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

a.insert(100, atIndex: 2)
print(a)
```

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

```
mutating func removeAtIndex(_ index: Int) -> Element
```

Metod diziden atılan elemanı bize verir. Örneğin:

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

a.removeAtIndex(1)
print(a)
```

Yapının `removeAll` metodu tüm elemanları silmek için kullanılır:

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

Örneğin:

```
var a: [Int] = [1, 2, 3, 4, 5, 6, 7, 8]
print("\(a.count), \(a.capacity)")
a.removeAll(keepCapacity: 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:

```
mutating func reserveCapacity(_ minimumCapacity: Int)
```

Örneğin:

```
var a = [Int]()

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

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

`Array<T>` bir yapı olduğu için değer türlerine ilişkindir. Yani `Array<T>` türünden bir değişken dizi elemanlarının kendisini tutar. 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. Örneğin:

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

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

    b[0] = 100
    b[1] = 200
}

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

Peki dizilerin bu biçimde fonksiyonlara aktarılması performans kaybına yol açmaz mı? İşte eğer biz dizide değişiklik yapmıyorsak, örneğin parametre değişkeni olan dizi `let` ile bildirildiyse (default durumda zaten böyledir) derleyici arka planda bir optimizasyonla diziyi zaten adres yoluyla fonksiyona aktarmaktadır. Tabi dizi `var` ile bildirilmişse ve fonksiyon içerisinde dizi elemanları değiştirilmişse bu durumda derleyici diziyi kopyalama

yöntemiyle fonksiyona aktarır. Aslında `Array<T>` yapısı "copy-on-write" özelliğine uygun bir biçimde tasarlanmış olabilir. Yani belki de derleyici dizileri hep adres yoluyla aktarmaktadır. Ancak dizi elemanı ilk değiştirildiğinde onun kopyasını çıkarmaktadır.

Tabii Swift'te bir nesneyi adres yoluyla da fonksiyonlara aktarmak mümkündür. Örneğin biz gerçekten fonksiyon içerisinde yaptığımız değişikliğin asıl diziyi etkilemesini isteyebiliriz (sort işlemi yapan bir fonksiyon düşününüz). Bunun için inout belirleyicisi kullanılır. Bu konu ileride ele alınacaktır. Örneğin:

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

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

    b[0] = 100
    b[1] = 200
}

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

Aynı türden iki dizi + operatörü ile toplama işlemine sokulabilir. Bu durumda yeni bir dizi nesnesi elde edilir. Bu yeni nesnenin elemanları soldaki dizinin sonuna sağdaki dizinin elemanlarının eklenmiş halidir. Örneğin:

```
var a: [Int] = [1, 2, 3, 4, 5]
var b: [Int] = [6, 7, 8, 9, 10]
var c: [Int]

c = a + b
print(c)      // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Benzer biçimde diziler için += işlemi de uygulanabilir. Zaten `a = a + b` her zaman `a += b` ile eşdeğerdir. Örneğin:

```
var a: [Int] = [1, 2, 3, 4, 5]
var b: [Int] = [6, 7, 8, 9, 10]

a += b
print(a)
```

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

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

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

Genel olarak bu biçimdeki, bir dizinin elemanlarına iki köşeli parantez `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.

Swift'te Tuple'lar

Tuple'lar özellikle fonksiyonel dillerde çok karşılaştığımız türlerdendir. Bir tuple bir ya da birden fazla değişkenin oluşturduğu topluluktur. Tuple'lar kullanımı kolaylaştırılmış yapılara benzetilebilir. Tuple'lar tipik olarak parantezler içerisinde eleman türleri belirtilerek bildirilir. Örneğin:

```
var a: (Int, Double)
```

Burada `a` ilk elemanı `Int`, ikinci elemanı `Double` türden bir tuple'dır. Tuple'a değer atama parantezler içerisinde değer listesi belirtilerek yapılır. Örneğin:

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

```
t = (100, 2.4)
print(t)
```

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

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

```
t = (100, 2.4)
print(t.0)           // 100
print(t.1)           // 2.4
```

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

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

```
pt = (10.2, 4.5)
print(pt.x)
print(pt.y)
```

Örneğin:

```
var per: (name: String, no: Int)
per = ("Ali Serçe", 123)
print("\(per.name), \(per.no)")
```

Tuple elemanlarının türleri belirtilmeyebilir. Bu durumda derleyici verilen ilk değerlerden tuple elemanlarının türlerini tespit eder. Örneğin:

```
var a = (10, 20.2)
print(a.dynamicType)           // (Int, Double)
```


Tuple elemanlarına isim ilkdeğer verilirken de verilebilir. Örneğin:

```
var a = (x: 10.2, y: 4.5)
print(a.x)
print(a.y)
```

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

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

a = (10.2, 3.2)
b = a          // geçerli
```

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

a = b          // geçerli
```

Tuple 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. İsimli elemanlara sahip tuple'larla isimsiz elemanlara sahip tuple'lar tür uyumu sağlandıktan sonra her zaman birbirlerine atanabilir. İsimli bir tuple isimsiz bir tuple'a atandığında artık atamanın yapıldığı hedef tuple isimsiz kabul edilir. Örneğin:

```
var pt1: (x: Int, y: Int) = (10, 20)
var pt2: (Int, Int)

pt2 = pt1

print("\(pt1.x), \(pt1.y)")
```

```
print("\(pt2.x), \(pt2.y)")    // error
print("\(pt2.0), \(pt2.1)")    // geçerli
```

Ancak atama işlemi bildirim ile ve tür belirtilmeden yapılıyorsa bu durumda eleman isimleri de atanana tarafa aktarılmaktadır. Örneğin:

```
var pt1: (x: Int, y: Int) = (10, 20)
var pt2 = pt1

print("\(pt1.x), \(pt1.y)")
print("\(pt2.x), \(pt2.y)")    // geçerli
```

Tuple'ların belli elemanlarına isim verip belli elemanlarına isim vermeyebiliriz. Örneğin:

```
var rect: (Int, Int, width: Int, height: Int) = (10, 20, 100, 100)
```

Tabii tuple'ın elemanına isim vermiş olsak bile biz yine o elemanı indeksli bir biçimde (yani numarayla) kullanabiliriz.

Tuple elemanları let ile bildirilirse elemanların değerleri değiştirilemez. Örneğin:

```
let a = (10, 20.2)
a.0 = 20    // error
a.1 = 12.3   // error
```

Bir tuple 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: (Int, Int) = (10, 20)
var x: Int, y: Int

(x, y) = pt    // geçerli
```

Ayrıştırma işleminde bazı elemanlar istenmiyorsa onun yerine '_' karakteri konulur. Örneğin:

```
var rect: (x: Int, y: Int, width: Int, height: Int) = (10, 20, 100, 100)
var (x, y, _, _) = rect
```

Şüphesiz tuple'ın elemanları başka bir tuple türünden olabilir. Örneğin:

```
var rect: (pt: (x: Int, y: Int), width: Int, height: Int) = ((10, 20), 100, 200)

print("x = \(rect.pt.x), y = \(rect.pt.y), width = \(rect.width), \(rect.height)")
```

Tuple'lar 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: Double, b: Double, c: Double

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

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

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

var result = getRoots(a, b, c)
if result == nil {
    print("gerçek kök yok")
}
else {
    print("x1 = \(result!.0), x2 = \(result!.1)")
}
```

String Yapısı

Yazıları temsil etmekte kullanılan String yapısı çok kullanıldığı için ayrı bir başlıkta ele alınması uygun görülmüştür. Bir String nesnesi String yapısının başlangıç matotlarıyla yaratılabilir. Örneğin:

```
var s: String = String() // s boş bir string
```

Örneğin:

```
let ch: Character = "a"
var s: String = String(ch)
```

Burada Character parametrelili init metodu kullanılmıştır. Örneğin:

```
var s: String = String(count: 10, repeatedValue: Character("a"))
print(s)
```

Diğer init metotları için Swift dokümanlarını inceleyebilirsiniz.

String yapısının kendisi diğer dillerin aksine bir collection özelliği göstermez. Yapının characters isimli property'si CharacterView isimli bir yapı türündendir. CharacterView yapısı String yapısının içerisinde bildirilmiştir, string'in karakterlerini dışarıya vermek için kullanılır. String.CharacterView yapısı CollectionType, SequenceType ve Indexable protokollerini (arayüzlerini) desteklemektedir. SequenceType yapısını destekleyen sınıf ve yapıların for in deyimiyle kullanılabildiğini anımsayınız. Bu nedenle CharacterView yapısı for in deyimiyle dolaşılabilir. Örneğin:

```
var s: String = "Test"

for ch in s.characters {
    print(ch, terminator: "")
}
print("")
```

String yapısının String.CharacterView.Index ve Range<String.CharacterView.Index> parametrelili iki "subscript" metodu vardır. Bu yüzden String yapısının herhangi bir elemanına köşeli parantez operatörü ile erişilebilir. Ancak köşeli parantezler içerisine Int değerler veremeyiz. Index yapısı CharacterView yapısı içerisinde bildirilmiştir ve elemana erişmekte kullanılan ara bir yapıdır. String yapısının startIndex property'si ilk karakter için Index değerini, endIndex property'si ise son karakterden bir sonrası için Index değerlerini bize verir. Örneğin:

```
var ch1, ch2: Character

ch1 = s[s.startIndex]
ch2 = s[s.endIndex]           // exception oluşur!
```

Index yapısının advancedBy metodu indeksi ilerletmek için kullanılmaktadır. Örneğin:

```
for var index = s.startIndex; index != s.endIndex; index = index.advancedBy(1) {
    print(s[index], terminator: "")
}
print("")
```

Görüldüğü gibi Index yapısının advancedBy metodu bize artırılmış ya da eksiltilmiş yeni Index nesnesini verir. Yani bizim bu metodun geri dönüş değerini kullanmamız gerekir. Örneğin bir yazının n'inci karakterine şöyle erişebiliriz:

```
var s = "eskişehir"

print(s[s.startIndex.advancedBy(4)])
```

Index yapısının ++ operatör metodu da vardır. Aynı işlem şöyle de yapılabilirdi:

```
for var index = s.startIndex; index != s endIndex; ++index {
    print(s[index], terminator: "")
}
print("")
```

Yazıyı şöyle tersten de yazdırabiliriz:

```
var s = "eskişehir"
var index = s endIndex

repeat {
    index = index.advancedBy(-1)
    print(s[index], terminator: "")
} while (index != s.startIndex)
```

Index yapısının predecessor metodu bir önceki indeks değerini elde etmek için successor metodu bir sonraki indeks değerini elde etmek için kullanılabilir. (Tabi aslında bu iki işlemi biz zaten advancedBy metodu ile de yapabiliriz.)

Swift'te tıpkı Objective-C ve C#'ta olduğu gibi extension metotlar vardır. Biz daha rahat çalışmak için String sınıfına aşağıdaki gibi subscript metotları ekleyebiliriz:

```
extension String {
    subscript (i: Int) -> Character {
        return self[self.startIndex.advancedBy(i)]
    }

    subscript (r: Range<Int>) -> String {
        let start = startIndex.advancedBy(r.startIndex)
        let end = start.advancedBy(r endIndex - r.startIndex)
        return self[Range(start: start, end: end)]
    }
}
```

Subscript metotlar ve extension metotlar ileride ele alınacaktır.

String'in karakterlerini bir diziye dönüştürüp erişimi dizi yoluyla da yapabiliriz. Örneğin:

```
var a:[Character] = [Character](s.characters)
for ch in a {
    print(ch, terminator: "")
}
print("")
```

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.

String yapısının append metodu ile yazının sonuna karakter ekleyebiliriz:

```
var s: String = "eskişehi"  
s.append(Character("r"))  
print(s)
```

appendContentsOf metodu ile bir yazının sonuna başka bir yazı eklenebilir:

```
var s: String = "eski"  
s.appendContentsOf("şehir")  
print(s)
```

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, C++'ın string sınıfına oldukça benzemektedir. Swift'teki String yapısını bir çeşit dizi gibi düşünebilirsiniz. Gerçekten de String yapısının içerisinde de bir kapasite tutulmaktadır. String'e eklama yapıldığında bu kapasite duruma göre artırılmaktadır.

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")  
}
```

String yapısının insert metodu bir karakteri belli indekse eklemek için insertContentsOf metodu bir yazıyı belli bir indekse eklemek için kullanılmaktadır:

```
mutating func insert(_ newElement: Character, atIndex i: Index)  
mutating func insertContentsOf<S : CollectionType where S.Iterator.Element == Character>(_  
newElements: S, at i: Index)
```

Örneğin yazının 4'üncü indeksine bir karakter eklemek isteyelim:

```
var s = "eskişehir"  
  
s.insert("x", atIndex: s.startIndex.advancedBy(4))  
print(s)
```

Örneğin:

```
var s = "esehir"  
var k = "kiş"
```

```
s.insertContentsOf(k.characters, at: s.startIndex.advancedBy(2))
print(s)
```

String yapısının removeAll metodu tüm karakterleri silmek için, removeAtIndex metodu belli bir indeksteki karakteri silmek için, RemoveRange ise belli bir aralıktaki karakterleri silmek için kullanılmaktadır.

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.appendContentsOf("izmir")
print(s)          // ankaraizmir
print(k)          // ankara
```

İki tırnak içerisinde önce bir ters bölü sonra ona yapışık biçimde parantezler içerisinde bir ifade yazılırsa Swift derleyicisi o ifadenin sayısal değerini hesaplayıp onu yazıya dönüştürür ve yazının o kısmına yerleştirir. Buna Swift'te "string interpolation" denilmektedir. Örneğin:

```
var a = 123
var s = "number:\(a)"
print(s)
```

Parantezler içerisindeki ifade karmaşık olabilir. Örneğin:

```
var a = 123, b = 2
var s = "number:\(a + b * 10)"
print(s)
```

Swift'te Sözlükler (Dictionaries)

Sözlük anahtar-değer çiftlerinden oluşan bilgileri tutan ve anahtar verildiğinde onun değerine hızlı 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.

Swift'te sözlükler köşeli parantezler içerisinde ':' atomu ile ayrılan anahtar ve değer türleriyle temsil edilmektedir. Örneğin anahtar türü K ve değer türü V olan bir sözlük [K: V] biçiminde temsil edilir. [K: V] türü Swift'in standart kütüphanesindeki Dictionary generic yapısının Dictionary<K, V> açılımı ile eşdeğerdir.

Bir sözlük boş olarak aşağıdaki gibi yaratılabilir:

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

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

```
var a: [String: Int] = ["Eskişehir": 26, "İstanbul": 34, "Kocaeli": 41, "Konya": 43]
```

Aynı bildirim tür belirtmeden de aşağıdaki gibi yapılabilirdi:

```
var a: ["Eskişehir": 26, "İstanbul": 34, "Kocaeli": 41, "Konya": 43]
```

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

```
var a: Dictionary<String, Int> = ["Eskişehir": 26, "İstanbul": 34, "Kocaeli": 41, "Konya": 43]
```

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

```
var a = ["Eskişehir": 26, "İstanbul": 34, "Kocaeli": 41, "Konya": 43]

print(a.count)      // 4
```

Sözlük türünün en önemli kullanıma nedeni anahtar verildiğinde değeri hızlı bir biçimde (algoritmik yöntemlerle) elde etmektir. Anahtara karşı değerin elde edilmesi için Dictionary yapısının subscript elemanı kullanılır. Yapının tek parametrelili subscript elemanında biz köşeli parantezler içerisinde anahtarı vererek bu subscript bize değeri seçeneysel olarak verir. Örneğin:

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

if let v = d["Selamixxx"] {
    print(v)
}
else {
    print("anahtar yok!")
}
```

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: [String : Int] = [String: Int]()

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

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.

Aynı işlem Dictionary yapısının updateValue metodu ile de yapılabilir:

```
mutating func updateValue(_ value: Value, forKey key: Key) -> Value?
```

Metodun birinci parametresi değeri, ikinci parametresi anahtarı alır. Değer varsa onu günceller geri dönüş değeri olarak önceki değeri bize seçeneysel olarak verir. Değer yoksa onu ekler ve nil değerine geri döner. Örneğin:


```

if let oldVal = d.updateValue(999, forKey: "Selami") {
    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:

```

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

for k in d.keys {
    print(k)
}

for v in d.values {
    print(v)
}

```

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

Aslında biz bir sözlüğü de doğrudan for in yapısı ile dolaşabiliriz. Bu durumda her adımda biz (K, V) biçiminde bir tuple elde ederiz. Örneğin:

```

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

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

```

Eleman silmek için yapının removeAtIndex metodu kullanılır. Fakat bu metot bizden anahtarı alarak silmeyi yapmaz. Bizden indeks değerini ister. İndeks değeri yine Int türden değildir. Daha önce String yapısında anlatıldığı gibi yapının içerisinde bildirilmiş olan Index isimli bir yapı türündendir. Dictionary yapısının startIndex ve endIndex property'leri bize ilk elemanın ve sondan bir sonraki elemanın İndeks değerlerini verir. Örneğin sözlükteki ilk elemanı şöyle söylebiliriz:

```

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

d.removeAtIndex(d.startIndex)

for e in d {
    print("\(e)")
}

```

Yapının removeAll metodu sözlük içerisindeki tüm elemanları siler. İsteğe bağlı argümanla kapasite korumasının yapıp yapılmayacağı belirlenebilmektedir.

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: Property'ler ve Metotlar. Property'ler yapının veri elemanlarını (data'larını), metotlar da işlevleri oluşturmaktadır.

Bir yapı nesnesinin yaratılması Şöyle yapılmaktadır:

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

Yapı nesnesi yaratıldığında nesneye ilkdeğer vermek için başlangıç metodu (initializer) çağrılmaktadır. Swift'te başlangıç metotlarının 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 başlangıç metotlarının geri dönüş değerleri diye bir kavramı yoktur. 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()
```

```
rect.disp()
```

Görüldüğü gibi yapıların elemanlarına (property ve metotlarına) yine nokta operatörüyle erişilmektedir.

Swift'te yapı nesneleri yaratıldığında (tıpkı C#'ta olduğu gibi) onların bütün property'lerine (veri elemanlarına) 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 property'lerine 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 property'lere 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 metotlarının 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
}

var rect: Rect = Rect()    // geçerli
```

Fakat örneğin:

```
struct Rect {
    var x: Double = 0
    var y: Double = 0
    var width: Double
    var height: Double
}

var rect: Rect = Rect()    // error!
```

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
    var y: Double
    var width: Double
    var height: Double
}

var rect: Rect = Rect(x: 10, y: 10, width: 100, height: 200)    // geçerli
```

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 property bildirim sırasına göre girilmesi zorunludur. Örneğin biz nesneyi aşağıdaki gibi yaratamazdık:

```
var rect: Rect = Rect(y: 10, x: 10, width: 100, height: 200)    // error!
```

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. Ancak parametresiz init metodunu eğer yapının tüm elemanlarına bildirim sırasında ilkdeğer verilmişse yazmaktadır.

Swift'te init metotlarının ilk parametrelerinin dışsal isimleri default durumda '_' biçiminde değildir. (Global fonksiyonlarda ve diğer metotlarda böyle olduğunu anımsayınız.) Yani eğer biz init metotlarında dışsal isimler için '_' belirlemesi yapmamışsak her zaman ilk parametre de dahil olmak üzere dışsal isimleri argüman listesinde belirtmek zorundayız. Örneğin:

```
struct Point {
    var x: Double
    var y: Double

    init(x: Double, y: Double)
    {
        self.x = x
        self.y = y
    }

    mutating func set(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)    // default durumda her parametrenin dışsal ismi
pt.disp()
pt.set(30, y: 40)    // default durumda ilk parametrenin dışsal ismi '_' biçiminde
```

Swift'te yapının bir init metodu başka bir init metodu içerisinde çağrılabilir. 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 property'leri 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)")
    }

    mutating func foo()
    {
        x = 20              // error, let property'ye değer atanmış
    }
}

```

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
}

```

```

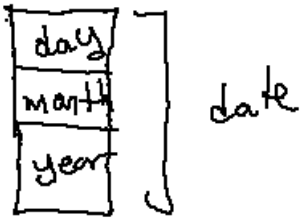
var month: Int
var year: Int

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)

```



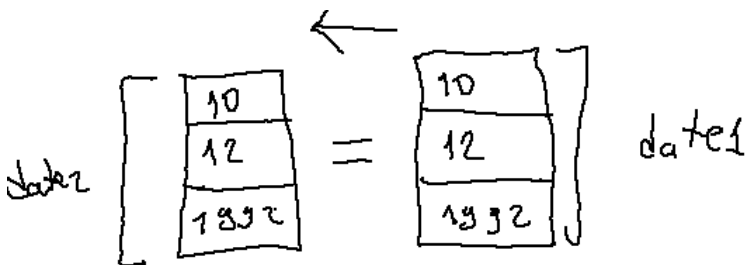
Şüphesiz derleyici yapı elemanlarını ardışıl bir biçimde tutar. Ancak dilde resmi olarak böyle bir garanti verilmemiştir.

Aynı türden iki yapı değişkeni birbirlerine atandığında onların bütün veri elemanlarının karşılıklı olarak birbirlerine atanacağına dikkat ediniz. 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. 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: 2015)
foo(endOfTheYear)

```

Yapıların bu biçimde fonksiyonlara ve metotlara parametre yoluyla aktarılması zaman kaybına yol açmaz mı? İşte Swift'te programcı bunun için fazlaca kaygılanmamalıdır. Çünkü derleyici yapı içerisinde nesnede bir değişiklik yapılmıyorsa zaten optimizasyon amacıyla yapıyı aslında adres yoluyla fonksiyona ya da metoda geçirir. Ancak yapıda değişiklik yapılıyorsa duruma göre bir performans problemi oluşabilir. Bu durumda programcı yapıyı inout anahtar sözcüğü ile adres yoluyla fonksiyona ya da metoda aktarmalıdır.

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.

Swift'te diğer nesne yönelimli dillerde olduğu gibi property'lerin veri elemanlarının gizlenmesi amacıyla kullanılmasına gerek yoktur. Zaten yapı ya da sınıfın veri elemanları değiştirildiğinde eski elemanla aynı isimli ve aynı türe sahip hesaplanmış property yazılabilmektedir.

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)
    }
}
```

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 niteliksiz olarak) kullanamazlar. Yani static olmayan metotlar (instance metotlardan) 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()
```

Statik metotlara Swift'te tür metotları (type methods) denilmektedir. Yapının tür metotları yine static anahtar sözcüğü ile bildirilir. 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: Int64) -> Bool
{
    return year % 400 == 0 || year % 4 == 0 && year % 100 != 0
}

func getTotalDays() -> Int64
{
    var total: Int64 = 0

    for var i: Int64 = 1900; i < Int64(year); ++i {
        total += Date.isLeapYear(i) ? 366 : 365
    }

    Date.monTab[1] = Date.isLeapYear(Int64(year)) ? 29 : 28

    for var i = 0; i < month - 1; ++i {
        total += Date.monTab[i]
    }
    total += day

    return total
}

func dateToText() -> String
{
    var totalDays = getTotalDays()

    return Date.dayText[Int(totalDays % 7)]
}
}

var today = Date(day: 23, month: 4, year: 1920)
var text = today.dateToText()
print(text)

```

Yapıların tür 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 dışsal isimlere sahip 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
    }

    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 kullanılması için o yapının 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 var i = 0; i < aw.count; ++i {
    print(aw[i], terminator: " ")
}
print("")

aw[0] = 100
aw[5] = 200
aw[9] = 300

for var i = 0; i < aw.count; ++i {
    print(aw[i], terminator: " ")
}
print("")

```

Subscript elemanların sentaks ve semantik özellikleri de hesaplanmış property'lerle aynı biçimdedir. Örneğin:

- 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(count: rowSize * colSize, repeatedValue: 0)
    }

    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 var i = 0; i < 3; ++i {
    for var k = 0; k < 3; ++k {
        im[i, k] = i + k
    }
}

for var i = 0; i < 3; ++i {
    for var k = 0; k < 3; ++k {
        print(im[i, k], terminator: " ")
    }
    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 değerın kendisini 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 birden fazla case yerleştirilemez.

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 değışkene 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.

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:

```

var d: Day

d = .Sunday              // geçerli
print(d)
d = .Thursday            // geçerli
print(d)

```

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 ilişkin olduğu tamsayı türleri vardır. Buna Swift terminolojisinde "temel değer türü (raw value type)" denilmektedir. 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 biz enum sabitlerine istediğimiz değerleri atayabiliriz. Diğer dillerde olduğu gibi enum sabitlerinin birine değer verilirse diğerleri onu izler. Örneğin:

```

enum Color : Int {
    case Red = 1, Green, Blue
}

```

Burada Color.Red = 1, Color.Green = 2, Color.Blue = 3 olacaktır. Ancak diğer değerlerin önceki değerleri izlemesi için listede sola doğru ilk kez değer verilen enum sabitine tamsayı bir değer verilmiş olması gerekir. Örneğin:

```

enum Direction : Double {
    case North = 1.5, East, South = 4, West = 3    // error 1.5 tamsayı değil
}

```

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 metot da içerebilir. Enum türlerinin başlangıç metotları (`init` metotları) söz konusu olabilir. Enum metotları içerisinde `self` anahtar sözcüğü yine enum değişkeninin kendisini temsil eder. 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()
```

Swift'te enum elemanlarına ayrı birer tür de atanabilir. Bu işlem enum isminden sonra parantezler içerisinde tür belirtilerek yapılır. Örneğin:

```
enum Fruit {
    case Apple(Int)
    case Banana(Double)
    case Cherry
}
```

Burada Fruit enum türünün Apple elemanı Int türüyle, Banana elemanı Double türüyle ilişkilidir. Cherry elemanı da hiçbir türle ilişkili değildir. Örneğin:

```
var a: Fruit = Fruit.Apple(10)
var b: Fruit = Fruit.Banana(12.4)
var c: Fruit = Fruit.Cherry
```

Burada bir noktaya dikkat çekmek istiyoruz: Enum türünün temel değer türü belirtildiğinde bu değer türü her elemana ilişkin kabul edilir. Oysa biz elemanlara ayrı ayrı değer türleri atayabilmekteyiz. Örneğin:

```
enum Fruit {
    case Apple(Int)
    case Banana(Int)
    case Cherry(Int)
}
```

bildirimi ile aşağıdaki bildiriminin işlevsel olarak birbirine yakındır:

```
enum Fruit : Int {
    case Apple
    case Banana
    case Cherry
}
```

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 eleman türü kadar yer ayırmaktadır. Enum elemanlarının temel değer türleri ne olursa olsun bu elemanların ilgili enum türünden olduğuna dikkat ediniz. Örneğin:

```
var info: Info // info ne kadar yer kaplayacak

info = Info.Name("test") // geçerli
print(info)

info = Info.IdentityNo(123) // geçerli
print(info)
```

Eğer enum elemanlarına tür bilgisi atadıysak artık rawValue property'si tanımlı değildir, biz rawValue property'sini kullanamayız. Yani rawValue property'si yalnızca enum'un kendisi tür bilgisiyle ilişkilendirilmişse anlamlıdır.

Hem enum'un kendisine hem de elemanlarına tür bilgisi atayamayız. Örneğin aşağıdaki enum bildirimi geçerli değildir:

```
enum Fruit : Int {           // error!
    case Apple(Int)
    case Banana(Double)
    case Cherry
}
```

Peki 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 önemli olan f'te ne olduğudur. O da örneğimizde Banana'dır. Maalesef Banana'ya atadığımız bu 12.4 değerini reflection dışında elde etmenin pratik bir yolu henüz yoktur. Bu değer Swift'te switch-case içerisinde elde edilip kullanılabilir. 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")
}
```

Buradaki case sentaksına dikkat ediniz. Enum elemanlarına iliştilen türler birer tuple da olabilir. 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")
}
```


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 metotların adresleri atanabilir.

Örneğin:

```
func foo(a: Int) -> Int
{
    return a * a
}

var f: (Int) -> Int
var result: Int

f = foo                // geçerli, foo'nun parametresi ve geri dönüşü 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
```

Swift'te fonksiyon türleri kategori olarak referans türlerine ilişkindir. Yani bir fonksiyon türünden değişken bir fonksiyonun adresini tutar. Yukarıdaki örnekte f 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.

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

```

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)
{
    var result: Int

```

```

    result = f(a)
    print(result)
}

func square(a: Int) -> Int
{
    return a * a
}

foo(10, f: square)

```

Örneğin:

```

func forEachString(strs: [String], _ f: (String) -> ())
{
    for str in strs {
        f(str)
    }
}

func disp(str: String)
{
    print(str)
}

let names = ["Ali", "Veli", "Selami", "Ayşe", "Fatma"]
forEachString(names, disp)

```

Array yapısının sort isimli metodu 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 diziye küçükten büyüğe sıraya dizeceksek ilk parametre ikinci parametreden küçükse true değerine, değilse false değerine geri dönmeliyiz. Örneğin:

```

let names = ["Ali", "Veli", "Selami", "Ayşe", "Fatma"]

func cmp(a: String, _ b: String) -> Bool
{
    return a < b
}

var result = names.sort(cmp)
print(result)

```

Örneğin:

```

struct Person {
    var name: String
    var no: Int
}

```

```

let 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)
]

func cmp1(a: Person, _ b: Person) -> Bool
{
    return a.name < b.name
}

func cmp2(a: Person, _ b: Person) -> Bool
{
    return a.no < b.no
}

var result = persons.sort(cmp1)
print(result)
print("-----")
result = persons.sort(cmp2)
print(result)

```

Array yapısının sort metodu MutableCollectionType arayüzünden gelmektedir. Böyle bir sort fonksiyonunu “kabarcık sıralaması (bubble sort)” algoritmasıyla aşağıdaki gibi yazabiliriz:

```

let names = ["Ali", "Veli", "Selami", "Ayşe", "Fatma"]
let numbers = [5, 7, 4, 2, 10]

func cmp(a: String, _ b: String) -> Bool
{
    return a < b
}

func cmp(a: Int, _ b: Int) -> Bool
{
    return a > b
}

func mysort<T>(array: [T], _ cmp: (T, T) -> Bool) -> [T]
{
    var sortedArray = array

    for var i = 0; i < sortedArray.count - 1; ++i {
        for var k = 0; k < sortedArray.count - i - 1; ++k {
            if !cmp(sortedArray[k], sortedArray[k + 1]) {
                let temp = sortedArray[k]
                sortedArray[k] = sortedArray[k + 1]
                sortedArray[k + 1] = temp
            }
        }
    }

    return sortedArray
}

var result1 = mysort(names, cmp)

```

```
print(result1)

var result2 = mysort(numbers, cmp)
print(result2)
```

Bir fonksiyonun 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
```

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
```

Bir fonksiyonun parametresi fonksiyon türündense biz onu çağırırken argüman olarak closure verebiliriz. Örneğin:

```
func foo(val: Int, f: (Int) -> Int)
{
    print(f(val))
}

foo(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 çağırılırken doğrudan aynı türden closure verilmiştir. Örneğin:

```
let names = ["Ali", "Veli", "Selami", "Ayşe", "Fatma"]

var result = names.sort({(a: String, b: String) -> Bool in return a < b})
print(result)
```

in anahtar sözcüğünden sonra closure içerisine istenildiği kadar deyim yerleştirilebilir. Örneğin:

```
func foo(val: Int, f: (Int) -> Int)
{
    print(f(val))
}

foo(10, f: {
    (a: Int)->Int in
    print("Ok")
    if a > 0 {
        return a * a * a
    }
    else {
        return a * a
    }
}) // 1000
```

Closure parametreleri yine default olarak let biçimdedir. Ancak biz onu var biçiminde de bildirebiliriz. Bu durumda parametre değişkeni üzerinde değişiklik yapabiliriz. Örneğin:

```
var f: (Int) -> Int
f = {(var a: Int) -> Int in a = a * 2; return a } // a değiştirilebilir
print(f(10))
```

Eğer biz burada var anahtar sözcüğünü kaldırsaydık a'ya değer atama kısmında error oluşurdu.

Closure'larda fonksiyon parametre türleri hiç belirtilmeyebilir. Bu durumda closure nasıl bir fonksiyon türüne 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))
```

Parametre türü belirtilmiyorsa biz artık isimlerin önüne var ya da let anahtar sözcüklerini koyamayız. Default durum yine let biçimindedir.

Anahtar Notlar: Her ne kadar Swift'in resmi gramerinde geçerli değilse de Swift derleyicileri parantezli biçimde tür belirtmeden var ya da let belirleyicilerini kabul etmektedir. Yani aşağıdaki closure Swift'in gramerine göre hatalı olmasına karşın Swift derleyicisi tarafından geçerli kabul edilmektedir:

```
var f: (Int, Int) -> Int
f = {(var a, let b) -> Int in a = a + b; return a }
print(f(10, 20))
```

Benzer biçimde closure'ların atandığı fonksiyon türlerinin geri dönüş değerlerinin türleri de bilindiği için closure'ların geri dönüş değerlerinin türlerinin 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:

```
let names = ["Ali", "Veli", "Selami", "Ayşe", "Fatma"]
let result = names.sort({a, b in return a < b}) // a ve b String türünden, geri dönüş değeri Bool türden
print(result)
```

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:

```
let names = ["Ali", "Veli", "Selami", "Ayşe", "Fatma"]

let result = names.sort({a, b in a < b})    // return anahtar sözcüğüne gerek yok
print(result)
```

Tabii closure'ın atandığı fonksiyon değişkeninin 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, return'e gerek yok
print(f(10, 20))
```

Burada parametre isimleri belirtilmediği için in anahtar sözcüğü de kullanılamaz. Örneğin:

```
var f: () -> ()

f = { print("test") }
f()
```

Türün let ya da var bildirimleriyle otomatik belirlendiği durumda closure'da tek bir ifade varsa derleyici bu ifadenin türüne bakarak closure'ın geri dönüş değerinin türünü tespit eder. Örneğin:

```
let f = {print("test")}
print(f.dynamicType) // () -> () çünkü print'in geri dönüş değeri yok
```

Fakat örneğin:

```
func foo() -> Int
{
    return 100
}

let f = {foo()}
print(f.dynamicType) // () -> Int çünkü foo'nun geri dönüş değerinin türü Int
```

Tabii closure'ın parametre türleri ve geri dönüş değerleri tam olarak belirtilmişse hedef türün tespiti tam olarak yapılabilir. Örneğin:

```
let f = {(a: Int) -> Double in return Double(a) * 3.14}
print(f.dynamicType) // (Int) -> Double
```

Eğer closure'da birden fazla ifade varsa ve return kullanılmamışsa geri dönüş değeri her zaman Void kabul edilir. Örneğin:

```
func foo() -> Int
{
    return 100
}

let f = {print("test"); foo()}
print(f.dynamicType) // () -> () çünkü closure'da tek bir ifade yok
```

Tabii -parametre isimleri ihmal edilsin ya da edilmesin- closure'da tek bir ifade yoksa ve fonksiyonun geri dönüş değeri varsa return kullanılması zorunludur. Örneğin:

```
var f: (Int, Int) -> Int

f = { if $0 > $1 { return $0 + $1 } else { return $0 * $1 }}
print(f(100, 20))
```

Swift'te eğer bir fonksiyon ya da bir metodun son parametresi bir fonksiyon türündense biz o fonksiyonu ya da metodu son argümanı closure olacak biçimde çağırıcaksak bu durumda closure içeriğini parametre parantezinden sonra açılan blok içerisinde 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 çağırırız:

```
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
}
```

Ya da örneğin:

```
foo(10) {
    $0 * $0
}
```

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. Örneğin:

```
func foo()
{
    var i = 20

    let f: () -> Void = { i *= 2 }

    print(i)    // 20
    f()
    print(i)    // 40
}

foo()
foo()
```

Capture edilen değişkenler faaliyet alanını kaybetse bile derleyici bunun kalıcılığını sağlamaktadır. Örneğin 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 metod bir fonksiyon türünden parametre değişkenine sahipse o parametreyi dışarda bir yere aktarmıyorsa derleyici optimizasyonu için parametre @noescape özelliği ile özniteliklendirilebilir. Örneğin:

```
func foo(@noescape f: () -> ())
{
    f()
}
```

```
foo({print("Ok")})
```

Tabi biz parametreyi dışarıda bir yere aktarıyorsak artık @noescape ile belirleme yapamayız. Örneğin:

```
var a: [() -> ()] = []

func foo(@noescape f: () -> ())
{
    a.append(f)    // error
}
```

Burada append fonksiyonunun parametresi @noescape ile nitelendirilmediği için çağrı error ile sonuçlanır. Fakat örneğin:

```
func bar(@noescape f: () -> ())
{
    f()
}

func foo(@noescape f: () -> ())
{
    bar(f)    // geçerli, bar da @noescape ile nitelendirilmiş
}

foo({print("Ok")})
```

Bir fonksiyon ya da metot () -> T türünden bir fonksiyon parametresine sahipse onu hiç küme parantezleri olmadan çağırabiliriz. Bunun için parametre değişkeninin @autoclosure özniteliği ile niteliklendirilmesi gerekir. Örneğin:

```
func foo(@autoclosure f: () -> Int)
{
    print(f())
}

foo(10 + 20)    // geçerli
```

Örneğin:

```
func foo(@autoclosure f: () -> Void)
{
    print(f())
}

foo(print("Ok"))    // geçerli
```

@autoclosure öznitelikleştirilmesi yapılmış bir fonksiyon ya da metodu fonksiyon ya da closure ile çağıramayız. Örneğin:

```
func foo(@autoclosure f: () -> Int)
{
    print(f())
}
```

```
foo({10 + 20})           // error
```

Swift'te yeni 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)
```

Örneğin:

```
typealias Proc = () -> ()

var f: Proc
f = { () -> () in print("ok")}    // f = { print("Ok")}
f()
```

Örneğin:

```
typealias Proc = () -> ()

var f: Proc
f = { Proc in print("ok")}      // f = { print("Ok")}
f()
```

Örneğin:

```
typealias Procs = [() -> ()]

var procs: Procs = [{() -> () in print("one")}, {() -> () in print("two")} ]
// var procs: Procs = [{print("one")}, {print("two")} ]

for var f in procs {
    f()
}
```

Sınıflar

Swift'te sınıf bildiriminin genel biçimi şöyledir:

```
class <sınıf ismi> {
    //...
}
```

Swift'te sınıflar yapıları oldukça benzemektedir. Benzerlikler şöyle özetlenebilir:

- Sınıflar da property elemanlara 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.

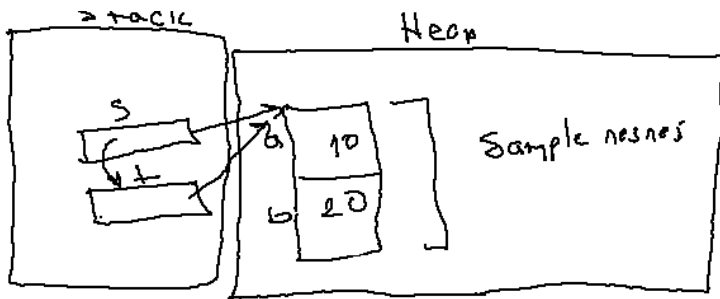
Sınıflar kategori olarak referans türlerine ilişkindir. Yani bir sınıf türünden bir değişken (referans) bildirdiğimizde o değişken sınıf nesnesinin adresini tutar. Örneğin:

```
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
    }
}
```

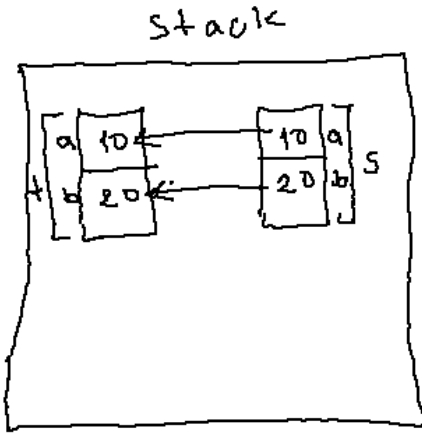
```

        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

```



Sınıflarda init metotları konusunda küçük bir farklılık vardır. Anımsanacağı gibi yapılarda init metodunun derleyici tarafından yazılma kuralı şöyledir:

- 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. Ancak parametresiz init metodunu eğer yapının tüm elemanlarına bildirim sırasında ilkdeğer verilmişse yazmaktadır.

Halbuki sınıflar için derleyicinin init metodunu derleyicinin kendisinin yazma 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.

Aradaki farka dikkat ediniz: Sınıflarda derleyici hiçbir biçimde eleman ilkdeğerlemesi yapan init metodunu yazmamaktadır.

Sınıfın property elemanları let ile bildirilmişse tıpkı var ile bildirildiğinde olduğu gibi ona ya bildirim sırasında ya da init metotları içerisinde ilkdeğer verme zorunluluğu vardır (yapılarda da böyleydi). Bunun dışında let property'lere başka yerde bir daha değer atanamaz. Ayrıca eğer let property'lerine bildirim sırasında ilkdeğer

verilmişse artık bunlara init metodu içerisinde de değer atanamamaktadır. Burada bir noktaya dikkat etmek gerekir. Sınıfın başka sınıf türünden property elemanı varsa property elemanının kendisi let durumdadır, onun gösterdiği yerdeki nesne let durumunda değildir. Örneğin:

```
class A {
    var x: Int

    init()
    {
        x = 10
    }
}

class B {
    let a: A = A()
}

var b: B = B()
b.a.x = 10      // geçerli
b.a = A()      // error
```

Burada B nesnesinin a isimli property elemanı let durumdadır. Yani biz a'nın kendisine artık başka bir değer atayamayız. a'nın gösterdiği yerdeki nesnenin elemanlarına atama yapabiliriz. A'nın bir yapı olması durumunda a property'sinin tamamı let durumdadır. Dolayısıyla biz onun x parçasına da değer atayamayız. Örneğin:

```
struct A {
    var x: Int

    init()
    {
        x = 10
    }
}

class B {
    let a: A = A()
}

var b: B = B()
b.a.x = 10      // error
```

Türetme İşlemleri

Anımsanacağı gibi Swift'te değer türleri (yani struct, enum türleri) türetmeye kapalıdır. Yani bir yapı ya da enum türünden biz bir sınıf, yapı ya da enum türü, bir sınıf türünden de bir yapı ya da enum türü türetemeyiz. Ancak sınıflardan sınıflar türetilebilir.

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")  
    }  
}
```

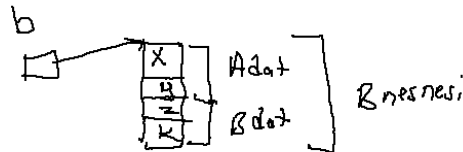
```
var b = B();
```

```
b.foo()    // geçerli  
b.bar()    // geçerli  
print(b.x) // geçerli  
print(b.y) // geçerli
```

Yine Swift'te de türemiş sınıf türünden bir nesne hem taban sınıf property'lerini (yani veri alanlarını) 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()
```

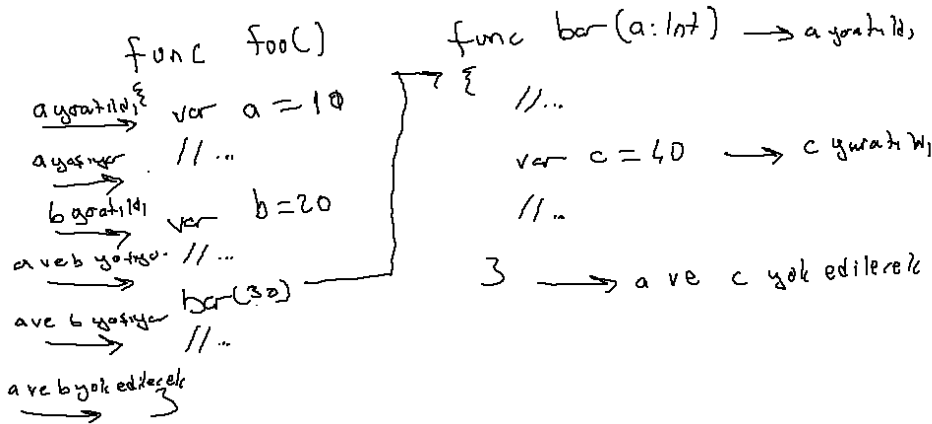


Yine Swift'te de türemiş sınıf nesnesinin taban sınıf kısmı ile türemiş sınıf kısmı ardışıl bir blok oluşturur. Nesnenin taban sınıf kısmı bellekte düşük adreste bulunmaktadır.

Programların Kod, Stack, Data ve Heap Alanları

Her uygulamanın bellek alanı kabaca dört kısma ayrılmaktadır: Kod, stack, data ve heap. Bir programdaki bütün global fonksiyonlar ve metotlar (yani yapıların ve sınıfların fonksiyonları) biraraya getirilerek kod bölümünde toplanmaktadır. Yani yapıların ya da sınıfların metotları nesnenin içerisinde değildir. Onlar uygulamanın adres alanının kod bölümündedir. Metotların sınıflarla ilişkisi mantıksal düzeydedir.

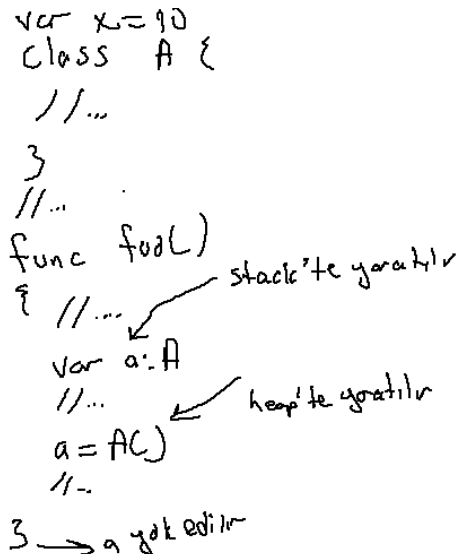
Fonksiyonların ve metotların parametre değişkenleri ve onların blokları içerisinde bildirilmiş olan yerel değişkenler stack'te yaratılmaktadır. Stack doldur-boşalt tarzı bir alandır. Bir değişken programın akışı fonksiyon ya da metotta o değişkenin bildirildiği yere geldiğinde stack'te yaratılır. Fonksiyon ya da metot bittiğinde stack'ten yok edilir. Örneğin:



Görüldüğü gibi bir fonksiyon ya da metot çağrılmadığı sürece onların içerisindeki yerel değişkenler zaten bellekte yer kaplıyor durumda değildir. Stack'te değişkenlerin yaratılması ve yok edilmesi çok hızlı bir biçimde yapılmaktadır.

Swift'te fonksiyonların ve sınıfların dışında bildirilen değişkenler uygulamanın data bölümünde yaratılırlar. Data bölümü programın yaşamı boyunca (yani run time boyunca) bellekte kalmaktadır.

Sınıf nesneleri heap'te yaratılır. Sınıf nesnelere ile onların adreslerinin tutulduğu değişkenleri (referansları) birbirine karıştırmayınız. Örneğin:



Görüldüğü gibi aşağıdaki gibi bir işlemde a (eğer fonksiyonun içerisindeyse) stack'te yaratılır. Ancak A nesnesi heap'te yaratılmaktadır:

```
var a = A()
```

Heap'te yaratılan sınıf nesneleri onu hiçbir referans göstermediği zaman sistem tarafından otomatik silinmektedir. Heap'teki nesnelerin yok edilmesi stack'e göre çok daha yavaş yapılmaktadır. 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) çağırır. 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 değeri almamış
    {
        self.y = y
    }
}

var b = B(y: 100);
```

Swift'te türemiş sınıfın taban sınıf kısmındaki property'lere ilkdeğerlerinin verilmesi için başvuru 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. 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.
- 3) 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)

```

4) Türemiş sınıfın "designated init metodu" ancak kendi sınıfının tüm property elemanlarına değer atandıktan sonra taban sınıfın "designated init" metodunu çağırabilir. Yani taban sınıfın init metodu çağırılmadan önce türemiş sınıfın tüm property 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 değerlendirilmektedir. Swift'te init metotları da override edildiği için böyle bir kurala gereksinim duyulmuştur.

5) Türemiş sınıfın "designated init" metodu taban sınıfın designated init metodunu çağırılmadan önce taban sınıfın property 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)
    }
}

```

6) Türemiş sınıfın "convenience init" metodu kendi sınıfının başka bir init metodunu çağırmadan önce kendi sınıfının ya da taban sınıfın bir property 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)
    }
}

```

7) 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

```

8) 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 x: Int

    init(x: Int)
    {
        self.x = x
    }

    convenience init()
    {
        self.init(x: 10)
    }
}

class B : A {
    var y: Int = 20

    override init(x: Int)
    {
        super.init(x: 10)
    }
}

var b: B = B() // geçerli

```

Swift'te init metotları da override edilebilir. Bu durum override işleminin ele alındığı kısımda açıklanmaktadır.

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 alınması için bunlar uydurulmuştur. Aslında örneğin aynı sorun C#'ta benzer biçimde çözülmüştür. Şöyle ki: 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")
}

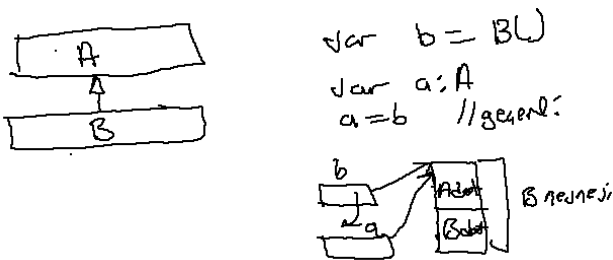
```

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

Nense yönelimli dillerin hepsinde türemiş sınıf türünden referans (ya da adres) taban sınıf türünden referansa 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 b: B = B(10, 20)
var a: A

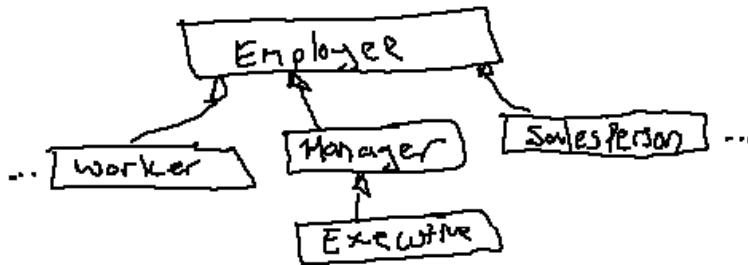
```

```

a = b
print(a.x)      // 10
a.x = 100
print(b.x)      // 100

```

Türemiş sınıf referansının taban sınıf referansına atanabilmesinin önemli sonuçları vardır. Bu sayede bir türetme şeması üzerinde genel işlemler yapan fonksiyonlar ya da metotlar yazılabilir. Örneğin:



```

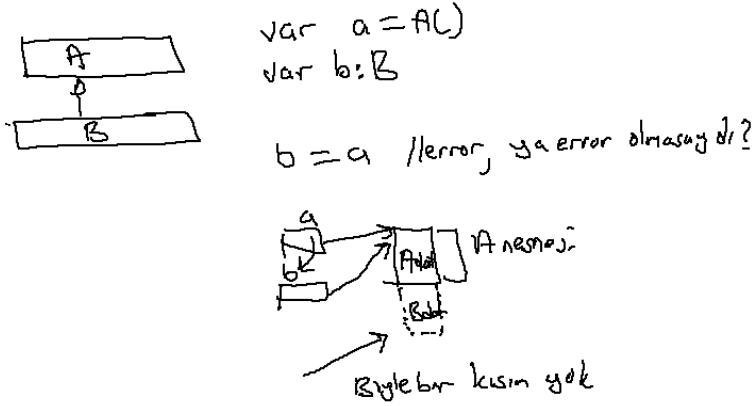
func displayInfo(e: Employee)
{
    //...
}

```

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 de gerekmektedir.

Yukarıdaki işlemin tersi yani taban sınıf referansının türemiş sınıf referansına atanması geçerli değildir. Eğer böyle bir atama yapılabilseydi 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:



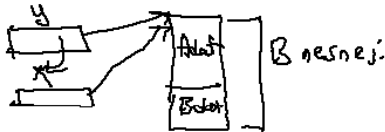
```
var x: A
var y: B = B()
```

```
x = y
```

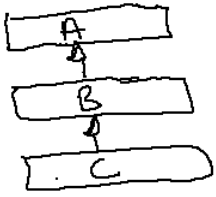
Burada x referansının static türü A'dır, dinamik türü B'dir. Çünkü x'in gösterdiği yerde bütünü B olan bir nesne vardır. x onun A parçasını göstermektedir.

```
var x: A
var y: B = B()
```

```
x = y
```



Burada y referansının statik türü de dinamik türü de B'dir. Örneğin:

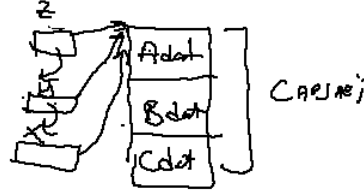


```

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'in ise hem statik hem de dinamik türü C'dir.

Referansın statik türü hiç değişmez. Fakat dinamik türü onun nereye gösterdiğine bağlı olarak değişebilir. Örneğin:

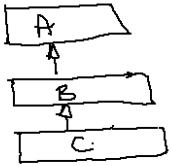
```

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

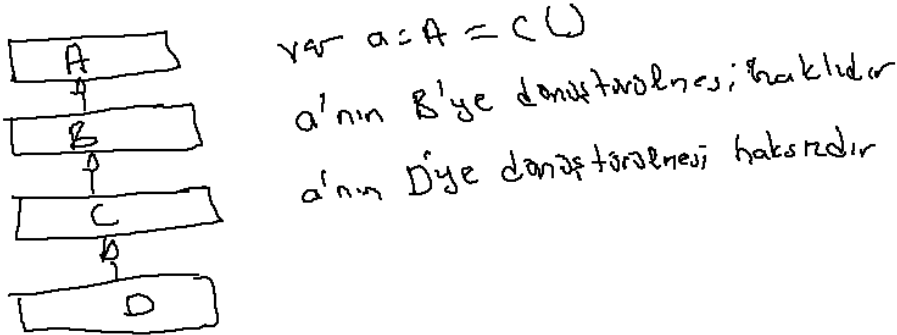
```

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 metod bazı gerekçelerden dolayı türmeiş 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:



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 as! ve as? operatörleriyle yapılmaktadır. Bu iki operatör de iki operandlı arae 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
```

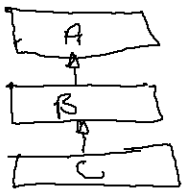
```

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! operatöyle aynı biçimde kullanılır. Ancak bu operatör dönüştürme haksızsa exception fırlatmaz nil referans üretir. Swift'te biz Java ve C#'ta olduğu gibi referanslara nil (onlarda nul) yerleştiremeyiz. Referanslara dahi Swift'te nil değerın yerleştirilebilmesi için o türün seçeneksel (optional) olması gerekir. Dolayısıyla as? operatörü de sağ tarafındaki tür (yani dönüştürülmek istenen tür) T ise bize T? türünden bir değer üretmektedir. Örneğin:



```

var a = A = B()
var c: C?
c = a as? C //geçerli
if c != nil {
    //... -> c unwrap yapılabilir
}
else {
    //...
}

```

Örneğin:

```
import Foundation
```

```

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)
var y: B
var z: C?

z = x as? C
if z != nil {
    print("\(z!.x), \(z!.y), \(z!.z)")
}
else {
    print("haksız dönüştürme!")
}

```

Swift'te (tıpkı Java ve C#'ta olduğu gibi) aralarında türetme ilişkisi olmayan iki sınıf arasında as! ya da as! operatörü ile 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 birşey 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 opeeratörün sol tarafındaki operand bir referans, sağ tarafındaki opereand 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 içeriyor sa true değeri, içermiyorsa false değeri üretir. Bu operatörün ürettiği değer Bool türündendir. Örneğin:

```

import Foundation

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 {
    var y = x as! B
    print("\(y.x), \(y.y)")
}
else {
    print("'x'in dinamik türü B'yi içermiyor")
}

```

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 metod 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 metod 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 func foo(val: Int) -> Int
    {
        print("C.foo")

        return val / a
    }
}

```

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 {
    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 fun 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 fun 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(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 çağrılmıştır. Örneğin:

```

import Foundation

class A {
    var a: Int

    init(a: Int)
    {
        self.a = a
    }

    fun 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 fun 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 fun foo(val: Int) -> Int
    {
        print("C.foo")

        return val / c
    }
}

fun 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 fonksiyonunun 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 metotları da override edilebilir. (Halbuki örneğin C++, Java ve C#'ta başlangıç metotları (constructors) override edilememektedir.) Örneğin:

```
class A {
    var a: Int

    init()
    {
        a = 0
    }
    //...
}

class B : A {
    var b: Int

    init()          // error!
    {
        b = 0

        super.init()
    }
    //...
}
```

Burada türemiş sınıfta taban sınıf ile aynı parametrik yapıya sahip init metodu kullanıldığı halde metodun başına override anahtar sözcüğü getirilmemiştir. Bildirim şöyle olması gerekirdi:

```
class A {
    var a: Int

    init()
    {
        a = 0
    }
    //...
}

class B : A {
    var b: Int

    override init()    // geçerli
    {
        b = 0

        super.init()
    }
    //...
}
```

Swift'te init metot 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.)

Yapılar türetmeye kapalı olduğu için yapıların init metotlarında required belirleyicisi kullanılamaz.

Swift'te "stored" ve "computed" property'ler override edilebilirler. (C#'ta da property'lerin override edilebildiğini anımsayınız.). 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 {
        get {
            return b * b * b
        }
        set {
            self.b = pow(newValue, 1.0 / 3.0)
        }
    }
}

let a: A = B(2.0, 3.0)
var result = a.val
print(result)           // 27.0

a.val = 64.0
result = a.val
print(result)           // 64

```

Burada val computed property'si override edilmiştir. Bu nedenle a.val ifadesinde a'nın dinamik türü B olduğu için B sınıfının val property'sinin get bölümü çalıştırılmıştır. Benzer durum set bölümü için de geçerlidir.

Swift'te read/write (yani mutable) bir property read-only olarak override edilemez. Yani başka bir deyişle property'nin hem get hem de set bölümü varsa biz onu override ederken hem get hem de set bölümünü bulundurmamak zorundayız. Ancak read-only bir 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.

Taban sınıftaki stored bir property (yani veri elemanı) türemiş sınıfta computed property olarak override edilebilir. Örneğin:

```

import Foundation

class A {
    var val: Double

    init(_ val: Double)
    {
        self.val = val
    }
}

class B : A {
    var b: Double

    init(_ a: Double, _ b: Double)
    {
        self.b = b
        super.init(a)
    }

    override var val: Double {
        get {

```



```

        return b * b * b
    }
    set {
        self.b = pow(newValue, 1.0 / 3.0)
    }
}

let a: A = B(2.0, 3.0)
var result = a.val
print(result)          // 27.0

a.val = 64.0
result = a.val
print(result)          // 64

```

Ancak taban sınıfın stored property'si let ile bildirilmişse (yani read-only ise) biz bunu türemiş sınıfta override edememekteyiz. Her ne kadar bunun mantıksal bir gerekçesi yoksa da mevcut Swift standardı buna izin vermemektedir.

Property gözlemcileri de override edilebilmektedir. Örneğin:

```

import Foundation

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)")
        }
    }
}

var a: A = B()

a.x = 10
print(a.x)

```

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. Bu işlem Java'da da aynı biçimde yapılmaktadır. Örneğin:

```
class A {
    //...
    final func foo()
    {
        //...
    }
}

class B : A {    // B sınıfında artık foo override edilemez
    //...
}
```

Bir sınıfın tamamı final yapılabilir (Java'da da böyle). 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 Java, C# ve C++'ta (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) 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 görmektedir. Ancak Swift tıpkı C++ gibi tepede bir sınıftan türetilmiş sınıflara sahip bir dil değildir. Fakat Swift'te de her türden referansı atayabileceğimiz Any isimli bir tür bulunmaktadır. Any türünden nesne yaratılamaz ancak referanslar bildirilebilir ve her türden referans any türünden referansa atanabilir. Örneğin:

```
import Foundation

class Sample {
    var a: Int

    init()
    {
        a = 0
    }

    init(a: Int)
    {
        self.a = a
    }
    //...
}

var s: Sample = Sample(a: 10)
```

```
var any: Any
```

```
any = s      // geçerli
```

Any türünden değişkene yalnızca referans türlerini değil yapı değişkenlerini de atayabiliriz. (Swift'te Int, String gibi türlerin yapı belirttiğini anımsayınız). Örneğin:

```
var any: Any // geçerli
```

```
var a = 123
```

```
any = a      // geçerli
```

Örneğin bir fonksiyonun parametresi Any türünden olabilir. Bu durumda biz fonksiyonu herhangi bir türden değişkenle çağırabiliriz:

```
func foo(a: Any)
```

```
{  
    //...  
}
```

```
var str = "ankara"
```

```
foo(str)
```

```
foo(123)
```

Swift'te Any türü Java ve C#'taki gibi bir taban sınıf değildir. Her türden değişkenin atanabildiği genel bir türdür.

Any türünden bir değişken 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.

```
var s = Sample(a: 10)
```

```
var a: Int = 20
```

```
var any: Any
```

```
var k: Sample
```

```
var b: Int
```

```
any = s      // geçerli
```

```
k = any as! Sample
```

```
print(k.a)
```

```
any = a
```

```
b = any as! Int
```

```
print(b)
```

Any türünden diziler de bildirilebilir. Bu dizler örneğin for deyimiyle dolaşılabilir:

```
var anys: [Any] = [10, 20, 30, "ali", "veli", "selami"]
```

```
for var any in anys {
```

```
    if any is Int {
```

```
        var val = any as! Int
```

```
        print(val)
```

```
    }
```

```
    else if any is String {
```

```
        var s = any as! String
```

```

        print(s)
    }
}

```

Peki any bir referans mıdır? Yanıt evet any bir referanstır. O halde biz any türünden bir referansa bir yapıyı 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 çıkarılır. Any referansı o kopyayı gösterir hale gelir. Any türünden referansa atama sonrasında biz asıl değişkeni değiştirirsek any referansının gösterdiği yerdeki kopya değişmeyecektir. 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

```

Any türünden referanslara fonksiyon türlerini de atayabiliriz. Örneğin:

```

var any: Any = {() -> () in print("Ok")}
var f = any as! () -> ()
f()

```

AnyObject türü Any türü gibidir. AnyObject ile Any arasındaki fark AnyObject türüne yalnızca kategori olarak referans türlerine ilişkin değişkenler atanabilirken Any türüne yapıların ve enum'ların da atanabilmesidir. Örneğin:

```

var any: AnyObject = Sample(a: 10)
var s = any as! Sample
print(s.a)

```

Anahtar Notlar: Her ne kadar Swift'in orijinal dokümanlarında AnyObject türünden bir referansa yalnızca kategori olarak referans türlerine ilişkin değişkenlerin atanabildiği belirtilmişse de mevcut Swift derleyicilerinde AnyObject türünden referansa yapı ve enum türünden değişkenlerin de atanabildiği görülmektedir. Tabii biz kursumuzda derleyiciyi de değil orijinal resmi dokümanları dikkate almaktayız.

AnyObject türünün kullanımıyla özellikle Cocoa ortamında sıkça karşılaşacağız.

Otomatik Referans Sayacı (Automatic Reference Counting) Mekanizması

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 mekanizmasıyla benzerdir. Yani her iki mekanizmanın da amacı heap'te artık kümsenin kullanmadığı nesnelerin otomatik serbest bırakılmasıdır. 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 lan bir nesneyi belli bir sayıda referans göstermektedir. Buna o nesnenin referans sayacı denilmektedir. 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 eksile de bilir. Nesnenin referans sayacı sıfıra geldiğinde artık nesneyi hiçbir referans göstermiyor durumdadır. Zaten bu noktadan sonra nesneyi bir referansın göstermesi de mümkün değildir. İşte Swift'te nesnenin referans sayacı sıfıra düşer düşmez ORS mekanizması yoluyla nesne heap'ten silinir. Örneğin:

```
class Sample {
    var a: Int

    init()
    {
        a = 0
    }

    init(a: Int)
    {
        self.a = a
    }
    //...
}

func bar(s: Sample)
{
    // RS = 3
    var m = s
    // RS = 4
}

func foo()
{
    var s = Sample() // RS'si izlenecek nesne

    // RS = 1

    var k = s

    // RS = 2

    bar(k)

    // RS = 2
}

// RS = 0

foo()

// RS = 0
```

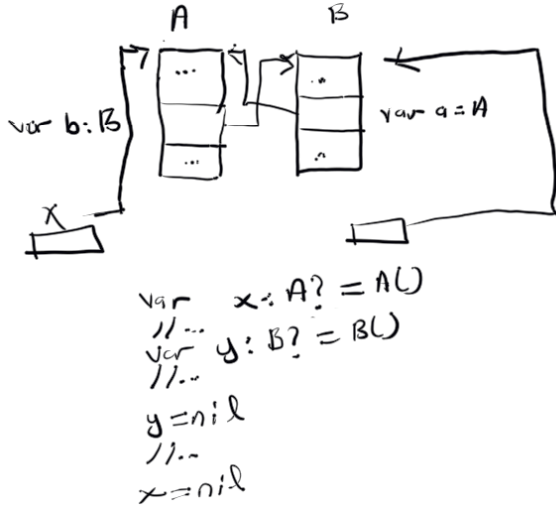
Bu örnekte foo çağrıldığında onun içerisinde yaratılan Sample nesnenin referans sayacı duruma göre artıp azalmıştır. foo metodunun çağrılması bitince o nesnenin referans sayacı sıfıra düşmüştür. İşte ORS mekanizması

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 onu gösteren referansa nil değeri yerleştirmektir. Tabii bu durumda türün de seçeneysel olması gerekmektedir. 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 sahipsiz (unowned) referans kavramları sokulmuştur.

weak bir referans seçeneysel 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
    }
}

class Country {
    var countryName: String
    var capital: City

    init(countryName: String, capitalName: String)
    {
        self.countryName = countryName
        capital = City(cityName: capitalName)
        capital.country = self
    }
}

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 ORS Country nesnesini yok edecektir. Böylece City nesnesini de gösteren referans kalmamış olacaktır. Dolayısıyla 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 referanslarla sahipsiz 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 edilince sistem zayıf referanslara nil değeri yerleştirir. Ancak aynı durum sahipsiz referanslar için geçerli değildir.

Sahipsiz referansları aşağıdaki gibi bir dönsüsellikte kullanabiliriz:

```
class City {
    var cityName: String
    unowned var country: Country

    init(cityName: String, country: Country)
    {
        self.cityName = cityName
        self.country = country
    }

    deinit {
        print("City deinit")
    }
}

class Country {
    var countryName: String
    var capital: City?
```



```

    init(countryName: String)
    {
        self.countryName = countryName
    }

    deinit {
        print("Country deinit")
    }
}

var country: Country? = Country(countryName: "Türkiye")
var capital = City(cityName: "Ankara", country: country!)
country = nil

```

Pekiye ne zaman zayıf referans ne zaman sahihsiz referans kullanmalıyız? Eğer referansımız seçeneksel olacaksa bu durumda mecburen zayıf referans, seçeneksel olmayacaksa sahihsiz 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:

```

import Foundation

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("test1")
s = nil
print("test2")
```

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 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 deinit metodu çağrılacaktır. Böylece biz Swift'te içeren nesnenin deinit metodu içerisinde içerilen nesne referansını kullanabiliriz. (Bunu Java ve C#'ta yapamayacağımızı anımsayınız).

Türetme durumunda türemiş sınıfın deinit metodu ana bloğun sonunda taban sınıfın deinit metodunu olarak çağırılmaktadır. Bunun için programcının birşey 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.

Türetme durumunda türemiş sınıfın deinit metodunun programcı tarafından yazılmadığını ancak taban sınıfın deinit metodunun yazılmış olduğunu varsayalım. 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.

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.

Eklenti bildiriminin genel biçimi şöyledir:

```
extension <isim> {  
    //...  
}
```

Burada isim eklenti yapılacağı 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. 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 exp = 1

        for var i = 0; i < index; ++i {
            exp *= 10
        }

        return self / exp % 10
    }
}

print(1234567[5])

```

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 (yani veri elemanı) içeremez. Protokoller metotlara ve hesaplanmış property'lere sahip olabilir. Protokol elemanları gövde içeremez. Örneğin:

```
protocol Test {
    func foo(a: Int) -> Int
    func bar()
}
```

Bir sınıf, yapı ya da enum protokolleri destekleyebilir. Bunun 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 destekleyiyorsa o protokülün bütün elemanlarını bildirmelidir. Yani örneğin protokolde bir metot varsa o protokolü destekleyen sınıf, yapı ya da enum o metodu gövdesiyle bildirmek zorundadır. Örneğin:

```
protocol Test {
    func foo(a: Int) -> Int
    func bar()
}

class Sample : Test {
    func foo(a: Int) -> Int {
        return a * a
    }

    func bar()
    {
        print("this is a test")
    }
    //...
}
```

Protokoldeki metotlar desteklenirken onların isimleri, parametre türleri, parametrelerin dışsal isimleri ve geri dönüş değerleri aynı biçimde olmak zorundadır.

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

    var count: Int {
        get {
            return 0
        }

        set {
            //...
        }
    }
}

```

Protokoldeki hesaplanmış property'ler 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()
    {
        //...
    }
}

```

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
    }
}
}

```

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 metot 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. Örneğin Sample sınıfı A protokolünü destekliyor olsun:

```

var a: A
var s = Sample(100)

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:

```

var a: A
var s = Sample()

a = s // geçerli
a.foo() // Sample.foo çağrılır

```

Örneğin:

```

import Foundation

protocol P {
    func operate(a: Int, b: Int) -> Int
}

class A : P {
    func operate(a: Int, b: Int) -> Int {
        return a + b
    }
}

class B : P {

```

```

    func operate(a: Int, b: Int) -> Int {
        return a * b
    }
}

class C : 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: b)
    print(result)
}

let a = A()
let b = B()
let c = C()

test(a, a: 20, b: 10)
test(b, a: 20, b: 10)
test(c, a: 20, b: 10)

```

Protokollerde çokbiçimliliğin doğal bir durum olduğunu 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 operate(a: Int, _ b: Int) -> Int
}

class A : P {
    func operate(a: Int, _ b: Int) -> Int {
        return a + b
    }
}

class B : A {
    //...
}

let p: P
let b = B()

p = b // geçerli
let result = p.operate(10, 20) // A.operate çağrılır
print(result)

```

Tabii türemiş sınıf taban sınıfın metodunu override edebilir. Bu durumda türemiş sınıftaki metot da çokbiçimli mekanizmaya dahil edilmiş olur. Örneğin:

```

import Foundation

protocol P {
    func operate(a: Int, _ b: Int) -> Int
}

class A : P {
    func operate(a: Int, _ b: Int) -> Int {
        return a + b
    }
}

class B : A {
    override func operate(a: Int, _ b: Int) -> Int {
        return a * b
    }
}

let p: P
let b = B()

p = b
let result = p.operate(10, 20) // geçerli
print(result)                 // B.operate 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()
    {
        //...
    }
}

class B : A, P { // geçerli
    func bar()
    {
        //...
    }
}

```

Bir protokol türünden bir collection 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:

```

import Foundation

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 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. 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:

```

import Foundation

protocol P {
    mutating func foo()
}

```

```

}

struct S : P {
    var a: Int

    mutating func foo()    // geçerli
    {
        //...
    }
}

class C : P {
    func foo()              // geçerli
    {
        //...
    }
}

```

Ayrıca protokollerdeki normal metotlar yapılarda ve enum'larda mutating olarak bildirilemez.

Protokoller init metotlarına sahip olabilir. Bu durumda o protokolü destekleyen sınıflarda ya da yapılarda o init metotlarının bulunuyor olması gerekir. Ancak böylesi bir durumda protokolü destekleyen sınıfta required anahtar sözcüğünün bulundurulması gerekir. Örneğin:

```

import Foundation

protocol P {
    func foo()
    func bar()
    init(a: Int)
}

class A : P {
    func foo()
    {
        //...
    }
    func bar()
    {
        //...
    }
    required init(a: Int) // required anahtar sözcüğü zorunlu
    {
        //...
    }
    //...
}

```

required belirleyicisi yapılarda ve enum'larda kullanılamadığı için init içeren protokolü bir yapı ya da enum destekliyse onların init bildirimlerinin önünde required belirleyicisi bulunmaz.

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:

```

import Foundation

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# ve Java'da bu özellik yok) bir protokolün yalnızca sınıflar tarafından desteklenmesi sağlanabilir. Bunun için protocol bildiriminde ':' atomundan sonra class anahtar söcüğü 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 `protocol<<protokol listesi>>`. (Yani `protocol` anahtar sözcüğünden sonra açılabilir parantezler içerisinde protokol istesi belirtilmelidir.) Örneğin:

```
import Foundation

protocol P1 {
    func foo()
}

protocol P2 {
    func bar()
}

func test(p: protocol<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)
```

Şüphesiz açılabilir parantezler içerisindeki protokol listesinin sırasının bir önemi yoktur. Örneğin:

```
let p1: protocol<P1, P2> = Sample()
let p2: protocol<P2, P1> = p1 // geçerli
```

Protokollerde Türetme

Bir protokol başka bir protokolden türetilebilir. (Burada "türetme" terimi kullanılmaktadır. "Destekleme" terimi kullanılmamaktadır.) Bu durumda türetilmiş protokol sanki tabanın protokolünün de elemanlarını içeriyormuş gibi bir etki söz konusu olur. Yani türetilmiş protokolü destekleyen bir sınıf, yapı ya da enum hem taban sınıfın hem de türetilmiş 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

Protokollerde de biz is operatörünü kullanabiliriz. Örneğin:

```

import Foundation

```

```

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)

```

Protoller de as? ve as! operatörlerinin sol taraf operandı olarak kullanılabilir. Başka bir deyişle biz bir protokolü bir sınıf, yapı ya da enum türüne şşşğıya 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()
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 protokülünü desteklemiyor olsa bile denetim derleme aşamasında yapılmaz. Programın çalışma zamanı sırasında yapılır. Tabi 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 protol 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:

```
import Foundation
```

```
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 aşamasındna 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:

```
import Foundation
```

```
protocol P {  
    func foo()  
}
```

```
//...
```

```

extension P {
    func bar()
    {
        print("P.bar")
    }
}

class Sample : P {
    func foo()
    {
        //...
    }
    //...
}

let s = Sample()

s.foo()      // geçerli
s.bar()      // geçerli

let p: P = s

p.foo()      // geçerli,
p.bar()      // geçerli

```

Anahtar Notlar: Swift'in orijinal dokümanlarında açıkça belirtilmemiş olsa da Swift derleyicisi ilgili sınıf ya da yapıda yeniden protokoldeki elemanı bildirdiğinde sorun çıkartmamaktadır. Bu durumda aşağıdan yani sınıf ya da yapı değişkeni yoluyla bu eleman kullanıldığında sınıf ya da yapının içerisinde bildirilmiş elemanın kullanıldığı varsayılmaktadır. Ancak protokol yoluyla kullanımda eklentide bildirilmiş olan elemanın kullanıldığı varsayılmaktadır.

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 türdenmiş gibi toplama, çıkartma vs. işlemlerine sokabiliriz. Bu durumda aslında arka planda bizim belirlediğimiz 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 fonksiyonlarına 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:

```
import Foundation

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(real: z1.real + z2.real, imag: z1.imag + z2.imag)

    return result
}

let z1 = Complex(real: 2, imag: 3)
let z2 = Complex(real: 3, imag: 4)
var result: Complex

result = z1 + z2
print(result.description)
```

Swift derleyicisi bir operatörle karşılaştığında önce operandların türlerine bakar. Eğer operand'lar temel türlerdence işlemi gerçekleştirir. Fakat operand'lardan en az biri bir sınıf ya da yapı türündence 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 edilir.

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)

```

++ ve -- operatörlerinin örnek ve sonek biçimleri yazılırken bu operatör fonksiyonlarının tek bir parametresi bulunmak zorundadır. Geri dönüş değerleri de aynı türde olacak biçimde oluşturulmalıdır. Operatörlerin hem örnek hem de sonek biçimlerinde artırma ya da eksiltme parametre üzerinde yapılmalıdır. Örnek biçimde parametrenin kendisine, sonek biçimde de artırılmadan öncek kopyaya geri dönülebilir:

```

prefix func -(a: Number) -> Number
{
    return Number(val: -a.val)
}

prefix func ++(a: Number) -> Number
{
    ++a.val

    return a
}

```

Swift'te diğer dillerin aksine operatör olmayan atom kümesinden de operatör fonksiyonu oluşturulabilir. Ancak operatör sembollerinin belli karakterlerden seçilmesi zorunlu tutulmuştur. Bu karakterleri 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 operator bildirimi ile onun örnek mi, aralık mi, sonek mi olduğunu, öncelik derecesini ve soldan-sağa, sağdan-solalılık durumunu (associativity) belirtmeliyiz. Bu işlemin genel biçimi şöyledir:

```

<durumu(infix, prefix, postfix)> operator <operatör sembolü>
{
    precedence <öncelik derecesi>
    associativity <left, right, none>
}

```

Aslında bizim standart operatör dediğimiz operatörler için de bu bildirimler önceden yapılmış durumdadır. Yani örneğin +, -, * / operatörleri için zaten yukarıdaki bildirimler yapılmış durumdadır. Pekiyi operatörün önceliği nasıl tespit edilmektedir? İşte operatör önceliği olarak bir sayı verilir. Standart operatörler için aşağıdaki öncelik değerleri belirlenmiştir:

<<	Bitwise left shift	None	Exponentiative, 160
>>	Bitwise right shift	None	Exponentiative, 160
*	Multiply	Left associative	Multiplicative, 150
/	Divide	Left associative	Multiplicative, 150

%	Remainder	Left associative	Multiplicative, 150
&*	Multiply, ignoring overflow	Left associative	Multiplicative, 150
&	Bitwise AND	Left associative	Multiplicative, 150
+	Add	Left associative	Additive, 140
−	Subtract	Left associative	Additive, 140
&+	Add with overflow	Left associative	Additive, 140
&−	Subtract with overflow	Left associative	Additive, 140
	Bitwise OR	Left associative	Additive, 140
^	Bitwise XOR	Left associative	Additive, 140
.. ..<	Half-open range	None	Range, 135
...	Closed range	None	Range, 135
is	Type check	Left associative	Cast, 132
as, as?, and as!	Type cast	Left associative	Cast, 132
??	Nil Coalescing	Right associative	Nil Coalescing, 131
<	Less than	None	Comparative, 130
<=	Less than or equal	None	Comparative, 130
>	Greater than	None	Comparative, 130
>=	Greater than or equal	None	Comparative, 130
==	Equal	None	Comparative, 130
!=	Not equal	None	Comparative, 130
===	Identical	None	Comparative, 130
!==	Not identical	None	Comparative, 130
~=	Pattern match	None	Comparative, 130
&&	Logical AND	Left associative	Conjunctive, 120
	Logical OR	Left associative	Disjunctive, 110
?:	Ternary Conditional	Right associative	Ternary Conditional, 100
=	Assign	Right associative	Assignment, 90
*=	Multiply and assign	Right associative	Assignment, 90
/=	Divide and assign	Right associative	Assignment, 90
%=	Remainder and assign	Right associative	Assignment, 90
+=	Add and assign	Right associative	Assignment, 90
-=	Subtract and assign	Right associative	Assignment, 90

<<=	Left bit shift and assign	Right associative	Assignment, 90
>>=	Right bit shift and assign	Right associative	Assignment, 90
&=	Bitwise AND and assign	Right associative	Assignment, 90
=	Bitwise OR and assign	Right associative	Assignment, 90
^=	Bitwise XOR and assign	Right associative	Assignment, 90
&&=	Logical AND and assign	Right associative	Assignment, 90
=	Logical OR and assign	Right associative	Assignment, 90

Öncelik dereceleri [0, 255] aralığında bir sayı olmak zorundadır. Biz kendi uydurduğumuz sembole yukarıdaki değerleri de dikkate alarak istediğimiz bir değeri verebiliriz. Örneğin biz Σ sembolünün önceliğine 145 verirse bunun önceliği +, - operatörlerinden daha fazla fakat * ve / operatörlerinden daha düşük olur.

Swift'te tek operandlı operatörler için öncelik derecesi ve associativity belirtilememektedir. Bunların öncelik derecesi her zaman 140 olarak değerlendirilir. Eğer "associativity" belirtilmezse default durum "none" kabul edilir. "None" associativity o operatörün başka bir operatörle kombine edilemeyeceğini belirtmektedir. Bu durumda önek bir operatör için bildirimi şöyle yapılabilir:

prefix operator Σ { }

Örneğin:

```
prefix operator  $\Sigma$  { }

prefix func  $\Sigma$ (a: [Int]) -> Int
{
    var total = 0

    for x in a {
        total += x
    }

    return total
}

let result =  $\Sigma$ [1, 2, 3, 4, 5] + 10

print(result)
```

infix operatör için de şöyle bir örnek düzenlenebilir:

```
infix operator  $\Sigma$ + {
    precedence 140
    associativity left
}
```

```

}

func Σ+(a: [Int], b: [Int]) -> [Int]
{
    let minCount = a.count < b.count ? a.count : b.count

    var result = [Int]()

    for var i = 0; i < minCount; ++i {
        result.append(a[i] + b[i])
    }

    return result
}

var result = [1, 2, 3] Σ+ [5, 7, 9, 10]
print(result)

```

Swift'te biz zaten tanımlı olan operatörler için de yeniden operatör fonksiyonları yazabiliriz. (C++'ta ve C#'ta bu mümkün değildir) Örneğin:

```

func +(a: Int, b: Int) -> Int
{
    return a * b
}

let x = 10 + 20 + 30
print(x)

```

Sınıflarda ve Yapılarda Erişim Belirleyici Anahtar Sözcükler

Bilindiği gibi Swift'te veri elemanlarına "stored property" denilmektedir. Aslında "stored property"ler derleyicinin kendisinin yazdığı get ve set metotları yoluyla erişimi yapmaktadır. Yani başka bir deyişle biz stored property'yi kullandığımızda aslında o değişkene zaten doğrudan erişmemekteyiz. O değişkene biz bir metot yoluyla erişmekteyiz. Örneğin:

```

class Sample {
    var no: Int

    init()
    {
        no = 0
    }
    //...
}

var s = Sample()
s.no = 100

```

Burada no isimli "stored property"ye erişim aslında arka planda "computed" bir property ile yapılmaktadır. Yani aslında yukarıdaki kodun derleyici tarafından derlenmiş biçimi şöyledir:

```

class Sample {
    var compilerGeneratedName: Int

```

```

var no: Int {
    get {
        return compilerGeneratedName
    }
    set {
        compilerGeneratedName = newValue
    }
}

init()
{
    compilerGeneratedName = 0
}
//...
}
var s = Sample()
s.no = 100

```

Görüldüğü gibi aslında C++, Java ve C#'ta olduğu gibi bizim veri elemanlarını private bölüme yerleştirerek onlara public metotlara erişmemiz işlemi Swift'te otomatik yapılmaktadır. Pekiyi sınıfın "stored property"lerinin isimleri ya da türleri değişirse onu kullanan kodların değiştirilmesi nasıl engellenecektir? İşte eğer biz sınıf ya da yapımızdaki "stored property'leri" değiştirirsek onlar için gerçekten bizim eski türü dikkate alarak "computed" property yazmamız gerekir. Örneğin:

```

class Sample {
    var otherNo: Double

    var no: Int {
        get {
            return Int(otherNo)
        }
        set {
            otherNo = Double(newValue)
        }
    }

    init()
    {
        otherNo = 0
    }
    //...
}
var s = Sample()
s.no = 100

```

Fakat yine de sınıfa eklenen isimlerin algılamayı kolaylaştırmak ve kapsülleme sağlamak için gizlenmesi gerekebilmektedir. İşte bu nedenle Swift'te 2.0 versiyonu ile private, public ve internal erişim belirleyicileri getirilmiştir.

internal default erişim belirleyicisidir. Yani erişim belirleyici anahtar sözcüklerden hiçbiri kullanılmamışsa 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. 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()
    {
        //...
    }
    //...
}
```

```
var s: Sample

s = Sample()

s.a = 10
```

Ayrıca Swift'te türlerin başına da erişim belirleyici anahtar sözcükler getirilebilmektedir. Burada da default durum `internal`'dir. `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 deyimin 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, 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 akışık else bloğunu bitirmesine izin verilmemiştir. Yani guard deyiminden sonraki deyimler ancak koşul doğruysa çalıştırılmaktadır.

guard deyimi daha çok seçeneksel türler için tercih edilmektedir. Tıpkı if deyiminde olduğu gibi guard deyiminde de otomatik unwrap özelliği ile nil karşılaştırması vardır. 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 seçeneksel biçiminin if deyiminin seçeneksel biçiminde 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
}
```

Burada if deyiminin dışında b'ye erişilemediğine dikkat ediniz. Örneğin:


```
var dict: [String: Int] = ["Ali": 123, "Veli": 234, "Selami": 543]

guard let val = dict["Veli"] else {
    print("value cannot find")
    return
}
```

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 çalıştırılır. 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 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 kaynağı 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 ErrorType isimli boş bir protokolü destekliyor olması gerekir. Örneğin:

```
class MyException : ErrorType {
    //....
}

enum YourException : ErrorType {
    //...
}
```

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 foo(a: Int) throws -> Int
{
    //...

    return 0
}
```

Swift'te C++, Java ve C#'ta olduğu gibi try bloğu yoktur. try deyimi yanına bir ifade almaktadır. Örneğin:

```
try foo()
```

gibi. Exception'ı yakalama do-catch bloklarıyla yapılır.

```
do {  
    //...  
}  
catch MyException.NotFound {  
    //...  
}  
  
catch MyException.IncorrectArgument {  
    //...  
}  
  
catch MyException.OutOfRange {  
    //...  
}
```

do bloğu içerisinde try deyimi yanına ifade yazılarak kullanılır. Bu durumda eğer o ifadede bir exception oluşursa akış do bloğunun aynı türden catch bloğuna aktarılır. O catch bloğu çalıştırıldıktan sonra diğer catch blokları atlanır ve akış catch bloklarının sonundan devam eder. Örneğin:

```
enum MyException : ErrorType {  
    case NotFound  
    case IncorrectArgument  
    case OutOfRange  
}  
  
func foo(a: Int) throws -> Int  
{  
    if a < 0 {  
        throw MyException.IncorrectArgument  
    }  
  
    print("Ok")  
  
    return 0  
}  
  
do {  
    let x = try foo(-20)  
    print(x)  
}  
  
catch MyException.IncorrectArgument {  
    print("argument must not be negative!")  
}
```

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 çağırılması zorunludur. Fakat try deyiminin do bloğu içerisinde çağırılması zorunlu tutulmamıştır. Ancak yakalanamayan exception'lar yine Java ve C#'ta olduğu gibi programın çökmesine yol açarlar.

Örneğin:

```
import Foundation

enum MyException : ErrorType {
    case NotFound
    case IncorrectArgument
    case OutOfRange
}

func foo(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
}

do {
    var no = try foo("Alice")
    print(no)
}

catch MyException.NotFound {
    print("Value not found")
}
```

catch cümlesinin 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 yapar bazı bilgiler de exception'ı yakalayacak catch cümlesine gönderilebilir. Bu örneğin enum elemanlarına tür bilgisi atayarak gerçekleştirilebilir. Örneğin:

```
enum MyException : ErrorType {
    case NotFound
    case IncorrectArgument
    case OutOfRange(String)
}

func foo(a: Double) throws -> Double
{
```

```

    guard a >= 0 else {
        throw MyException.OutOfRange("value must be positive or zero!")
    }

    return sqrt(a)
}

do {
    let result = try foo(-20)
    print(result)
}

catch MyException.OutOfRange(var msg) {
    print("Error: \(msg)")
}

```

Burada catch sentaksına dikkat ediniz: Enum'un parametresinde tür belirtilmemektedir.Yani bildirim aşağıdaki gibi yapılmamalıdır:

```

catch MyException.OutOfRange(var msg: String) {    // error!
    print("Error: \(msg)")
}

```

Tabii enum elemanı bir sınıf ya da yapı türünden de olabilir. Örneğin:

```

enum MyException : ErrorType {
    case NotFound
    case IncorrectArgument(IncorrectArgumentException)
    case OutOfRange
}

class IncorrectArgumentException {
    var msg: String

    init(msg: String)
    {
        self.msg = msg;
    }

    var description : String {
        return msg
    }
}

func foo(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 foo(-20)
    print(result)
}

catch MyException.IncorrectArgument(var 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 ErrorType referansı elde ederiz. Bu referansın dinamik türüne ilişkin bir kalıp oluşturarak catch düzenlemesini yapabiliriz. Örneğin:

```
class IncorrectArgumentException : ErrorType {
    var msg: String

    init(msg: String)
    {
        self.msg = msg;
    }

    var description : String {
        return msg
    }
}

func foo(a: Double) throws -> Double
{
    guard a >= 0 else {
        throw IncorrectArgumentException(msg: "value must be positive or zero!")
    }

    return sqrt(a)
}

do {
    let result = try foo(-20)
    print(result)
}

catch var e where e is IncorrectArgumentException {
    var 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 do bloğu içerisinde kullanılmasının da bir anlamı yoktur. Örneğin:

```
enum MyException : ErrorType {
    case IncorrectArgument(String)
```

```

}

func foo(a: Double) throws -> Double
{
    guard a >= 0 else {
        throw MyException.IncorrectArgument("value must be positive or zero!")
    }

    return sqrt(a)
}

let result: Double? = try? foo(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 eder. try! operatörünün de do bloğunun içerisinde kullanılmasının bir anlamı yoktur. Örneğin:

```

enum MyException : ErrorType {
    case IncorrectArgument(String)
}

func foo(a: Double) throws -> Double
{
    guard a >= 0 else {
        throw MyException.IncorrectArgument("value must be positive or zero!")
    }

    return sqrt(a)
}

let result: Double = try! foo(-20)
print(result)

```

Fonksiyonların inout Parametreleri

Normal olarak fonksiyonlar çağrılırken argümanlardan parametre değişkenlerine karşılıklı bir atama yapılır. Eğer fonksiyonun parametre değişkeni inout belirleyicisi ile bildirilmişse bu durumda ilgili argümanın adresi fonksiyona geçirilir. Artık fonksiyon içerisinde o parametre değişkenine atama yapıldığında parametreye karşılık gelen argümana atama yapılmış gibi bir etki oluşur. inout parametrelili bir fonksiyon aynı türden bir değişkenle çağrılmak zorundadır. Ayrıca argümanın & operatörü ile niteliklendirilmesi gerekir. Örneğin:

```

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

```



```

}

var x: Int = 10

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

```

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 Swift'te ilgili parametreye karşılık gelen argümanın & operatörü ile niteliklendirilmesiyle çağrılırlar. Örneğin:

```

func swap(inout a: Int, inout _ b: 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 birarada kullanılamamaktadır.

Generic'ler

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 var i = 1; i < a.count; ++i {
        if max < a[i] {
            max = a[i]
        }
    }

    return 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]) -> Double
{
    var max = a[0]

```

```

    for var i = 1; i < a.count; ++i {
        if max < a[i] {
            max = a[i]
        }
    }

    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 bildirilirler. Derleyici de o şablona bakarak ilgili türden fonksiyonu, sınıfı ya da yapıyı bizim için oluşturur.

Swift'teki generic'lerin bildirimi ve kullanımı C# ve Java'daki generic'lere oldukça benzemektedir.

Generic Fonksiyonlar ve Metotlar

Generic bir fonksiyonun bildirimi tıpkı C# ve Java'da olduğu gibi fonksiyon ya da metot isminden sonra açılacak 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>(inout a: T, inout b: T)
{
    let temp = a
    a = b
    b = temp
}

```

Buradaki T türü generic fonksiyon kullanılırken derleyici tarafından argümanlara bakılarak otomatik tespit edilir. Örneğin:

```

var x: Int = 10, y: Int = 20
swap(&x, &y)
print("x = \(x), y = \(y)")

```

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++ 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 metod 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 açısız 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 metodlarının da içerisinde) kullanılabilir. 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
    }
}
```

Generic bir tür kullanılırken tür isminden sonra açılabilir parantezler içerisinde kesinlikle tür argümanlarının belirtilmesi gerekir. Örneğin:

```
var stack = Stack<Int>()
for var i = 0; i < 10; ++i {
    stack.push(i)
}

while !stack.isEmpty {
    var elem = stack.pop()
    print("\(elem) ", terminator:"")
}
print("")
```

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 birarada bulunamazlar. Halbuki C++ buna izin vermektedir. C#'ta da overload işlemi kısmen yapılabilmektedir.

Generic'lerde Tür Kısıtları (Type Constraints)

Bir generic fonksiyon ya da sınıf generic tür parametreleri hangi türden açılırsa açılsın anlamlı olmak zorundadır. Aksi halde derleme aşamasında error oluşur. Örneğin:

```
func findIndex<T>(a: [T], val: T) -> Int?
{
    for var i = 0; i < a.count; ++i {
        if a[i] == val { // error!
            return i
        }
    }

    return nil
}
```

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 var i = 0; i < a.count; ++i {
        if a[i] == val { // geçerli, T'nin == operatör metodu olmak zorunda
            return i
        }
    }
}
```

```

    return nil
}

```

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:

```

struct Number {
    var val: Int = 0

    init(_ val: Int)
    {
        self.val = val
    }
}

var a = [Number(10), Number(20), Number(30), Number(40), Number(50)]
var index = findIndex(a, val: Number(30))    // error!
if index != nil {
    print(index!)
}

```

Equatable protokolü == operatör fonksiyonuna sahiptir. Yani Equatable protokolünü destekleyen bir sınıf ya da yapı == operatör fonksiyonuna sahip olmak zorundadır. Örneğin:

```

struct Number : Equatable {
    var val: Int = 0

    init(_ val: Int)
    {
        self.val = val
    }
}

func ==(a: Number, b: Number) -> Bool
{
    return a.val == b.val
}

var a = [Number(10), Number(20), Number(30), Number(40), Number(50)]    // geçerli
var index = findIndex(a, val: Number(30))
if index != nil {
    print(index!)
}

```

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 isim protokol 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. 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 Comparable protokolünü desteklemektedir. Örneğin:

```
var a = [23, 45, 28, 54, 98, 12]
var max = getMax(a)
print(max)
```

Örneğin:

```
struct Number : Comparable {
    var val: Int = 0

    init(_ val: Int)
    {
        self.val = val
    }
}

func ==(a: Number, b: Number) -> Bool
{
    return a.val == b.val
}

func <(a: Number, b: Number) -> Bool
{
    return a.val < b.val
}

func getMax<T: Comparable>(a: [T]) -> T
{
    var max = a[0]
```

```

    for i in 1..

```

Swift'te generic protokol kavramı yoktur. Ancak generic protokoller dolaylı bir biçimde belli düzeyde oluşturulabilmektedir. Bir protokolde typealias anahtar sözcüğü ile bir tür ismi uydurulursa protokolün elemanları ona dayalı olarak oluşturulabilmektedir. Örneğin:

```

protocol P {
    typealias T

    func foo(a: T) -> T
}

```

Buradaki typealias 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
    }
    //...
}

```

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 XXXLiteralConvertible isimli protokol vardır. Bu protokolleri destekleyen türlere XXX kategorisinden sabitler doğrudan atanabilir. (Swift'te sabitlerin türlerinin olmadığını yalnızca kategorilerinin olduğunu anımsayınız.) Örneğin IntegerLiteralConvertible protokolünü destekleyen bir türe biz nokta içermeyen bir sayıyı atayabiliriz. Örneğin:

```
class Sample : IntegerLiteralConvertible {
    //...
}

var s: Sample
s = 123    // geçerli
```

XXXLiteralConvertible protokolleri bir init metodu içermektedir. Bu init metodu XXX kategorisinden sabit değerleri alabilmektedir. Örneğin IntegerLiteralConvertible arayüzü aşağıdaki gibi oluşturulmuştur:

```
protocol IntegerLiteralConvertible {
    typealias IntegerLiteralValue
    init(integerLiteral value: IntegerLiteralType)
}
```

Örneğin:

```
class Sample : IntegerLiteralConvertible {
    var val: Int

    required init(integerLiteral val: Int)
    {
        self.val = val
    }
}

var s: Sample

s = 123    // geçerli, Sample(123) ile eşdeğer
```

Biz burada IntegerLiteralConvertible 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 nesne yaratmaktadır. Yani bu atama işlemi aslında yalnızca kısa bir yazım oluşturmaktadır. C++ ve C#'ta bu işlemin tür dönüştürme operatör fonksiyonlarıyla yapıldığını anımsayınız.