ATOMİK KOTLIN

Bruce Eckel

Svetlana Isakova

Çeviri ve Editöryal Katkı:
Hasan Tunçay

Atomic Kotlin (Türkçe Baskı)

Bruce Eckel, Svetlana Isakova ve Hasan Tunçay

Bu kitap https://leanpub.com/AtomicKotlin-tr adresinde satılmaktadır

Bu versiyon 2025-07-21 tarihinde yayınlandı

ISBN 978-0-9818725-4-4

Bu bir Leanpub kitabıdır. Leanpub, yazarlara ve yayıncılara Yalın Yayıncılık süreciyle güç kazandırır. Yalın Yayıncılık, okuyucu geri bildirimi almak için hafif araçlar ve birçok iterasyon kullanarak devam eden bir e-kitabı yayınlama, doğru kitaba ulaşana kadar yön değiştirme ve başardığınızda ivme kazanma sürecidir.

İçindekiler

Telif Hakkı

Atomic Kotlin

Bruce Eckel, MindView, LLC Başkanı ve Svetlana Isakova, JetBrains sro.

eKitap ISBN 978-0-9818725-4-4

Versiyon 1.0: Aralık 2020

Versiyon 1.1: Kasım 2021

Baskı Kitap ISBN 978-0-9818725-5-1

İlk baskı: Ocak 2021

İkinci baskı: Kasım 2021

Kasım 2021 güncellemeleri, Kotlin 1.5 için yapılan ayarlamaları ve düzeltmeleri içerir.

eKitap ISBN, Leanpub ve Stepik eKitap dağıtımlarını kapsar ve her ikisi de *www.AtomicKotlin.com* üzerinden mevcuttur.

Bu kitabı *www.AtomicKotlin.com* üzerinden satın alarak, devam eden bakım ve güncellemelerini destekleyin.

Crested Butte, Colorado, ABD ve Münih, Almanya'da oluşturulmuştur.

Metin Amerika Birleşik Devletleri'nde basılmıştır.

Kapak tasarımı Daniel Will-Harris tarafından yapılmıştır, www.Will-Harris.com[1]

Bizi ziyaret edin: *www.AtomicKotlin.com.*

Kaynak Kodu

Bu kitabın tüm kaynak kodları, Github[3] üzerinden dağıtılan telif hakları korunmuş ücretsiz yazılım olarak mevcuttur. En güncel sürüme sahip olduğunuzdan emin olmak için bu resmi kod dağıtım sitesidir. Bu kodu sınıflarda ve diğer eğitim durumlarında bu kitabı kaynak olarak belirtmek şartıyla kullanabilirsiniz.

Bu telif hakkının birincil amacı, kodun kaynağının doğru şekilde belirtilmesini sağlamak ve izinsiz olarak kodun yeniden yayımlanmasını önlemektir. (Bu kitap kaynak olarak belirtilmek kaydıyla, kitabın örneklerinin çoğu medyada kullanılması genellikle bir sorun teşkil etmez.)

Her kaynak kodu dosyasında aşağıdaki telif hakkı bildirimine bir referans bulacaksınız:

[1] http://www.Will-Harris.com
[2] https://kotlinlang.org/foundation/kotlin-foundation.html
[3] https://github.com/BruceEckel/AtomicKotlinExamples

Projelerinizde ve sınıfta (sunum materyalleriniz dahil) her kaynak dosyada görünen telif hakkı bildirimi korunduğu sürece kodu kullanabilirsiniz.

Bölüm I: Programlamanın Temelleri

Programlamada inanılmaz derecede cezbedici bir şey vardı—**Vint Cerf**

Bu bölüm programlamayı öğrenen okuyucular içindir. Eğer deneyimli bir programcıysanız, Özet 1 ve Özet 2'ye atlayabilirsiniz.

Giriş

> Bu kitap, adanmış yeni başlayanlar ve deneyimli programcılar içindir.

Eğer daha önce programlama bilgisine sahip değilseniz, ama "adanmış" iseniz, kendi başınıza çözebileceğiniz kadar bilgi veriyoruz. Bitirdiğinizde, programlama ve Kotlin konusunda sağlam bir temele sahip olacaksınız.

Eğer deneyimli bir programcıysanız, Özet 1 ve Özet 2'ye geçip oradan devam edebilirsiniz.

Kitabın başlığındaki "Atomik" kısmı, bölünemeyen en küçük birimler olan atomlara atıfta bulunur. Bu kitapta her bölümde sadece bir kavram tanıtmaya çalışıyoruz, bu yüzden bölümler daha fazla alt bölümlere ayrılmıyor—bu yüzden onlara *atom* diyoruz.

Kavramlar

Tüm programlama dilleri özelliklerden oluşur. Bu özellikleri sonuç üretmek için kullanırsınız. Kotlin güçlüdür—sadece zengin bir özellik setine sahip olmakla kalmaz, aynı zamanda bu özellikleri genellikle birçok şekilde ifade edebilirsiniz.

Eğer her şey size çok hızlı bir şekilde yüklenirse, Kotlin'in "çok karmaşık" olduğunu düşünebilirsiniz.

Bu kitap sizi bunaltmamayı amaçlar. Dili dikkatli ve kasıtlı bir şekilde öğretmek için aşağıdaki ilkeleri kullanırız:

1. **Bebek adımları ve küçük zaferler**. Bölüm zorbalığını terk ediyoruz. Bunun yerine, her küçük adımı bir *atomik kavram* veya basitçe *atom* olarak sunarız, ki bu küçük bir bölüm gibi görünür. Her atomda sadece bir yeni kavram sunmaya çalışırız. Tipik bir atom, bir veya daha fazla küçük, çalıştırılabilir kod parçası ve ürettiği çıktıyı içerir.

2. **İleriye yönelik referanslar yok**. Mümkün olduğunca, "Bu özellikler daha sonraki bir atomda açıklanmıştır" demekten kaçınırız.
3. **Diğer programlama dillerine referans yok**. Sadece gerektiğinde yaparız. Anlamadığınız bir dildeki bir özelliğe yapılan benzetme yardımcı olmaz.
4. **Göster, anlatma**. Bir özelliği sözlü olarak tanımlamak yerine, örnekler ve çıktılar kullanmayı tercih ederiz. Kodda bir özelliği görmek daha iyidir.
5. **Teoriden önce pratik**. Dili önce mekanik açıdan göstermeye çalışırız, sonra bu özelliklerin neden var olduğunu açıklarız. Bu, "geleneksel" öğretimden ters bir yaklaşımdır, ancak genellikle daha iyi çalışıyor gibi görünüyor.

Özellikleri bilirseniz, anlamını çözebilirsiniz. Genellikle bir sayfalık Kotlin'i anlamak, başka bir dildeki eşdeğer kodu anlamaktan daha kolaydır.

Dizin Nerede?

Bu kitap Markdown ile yazılmış ve Leanpub ile üretilmiştir. Maalesef, ne Markdown ne de Leanpub dizinleri desteklemiyor. Ancak, her atomda tek bir konu içeren en küçük bölümleri (atomlar) oluşturarak, içindekiler tablosu bir tür dizin işlevi görür. Ek olarak, eKitap sürümleri kitap boyunca elektronik arama yapmaya olanak tanır.

Çapraz Referanslar

Kitaptaki bir atoma referans şu şekilde görünür: Giriş, ki bu durumda mevcut atoma referans verir. Çeşitli eKitap formatlarında, bu o atoma bir hiperlink üretir.

Biçimlendirme

Bu kitapta:

- *İtalik* yeni bir terim veya kavramı tanıtır ve bazen bir fikri vurgular.
- `Sabit genişlikte yazı tipi` program anahtar kelimelerini, tanımlayıcıları ve dosya isimlerini belirtir. Kod örnekleri de bu yazı tipinde olup, kitabın eKitap sürümlerinde renklendirilmiştir.

- Düz yazıda, bir işlev adını boş parantezlerle takip ederiz, örneğin `func()`. Bu, okuyucuya bir işlevle karşı karşıya olduklarını hatırlatır.
- eKitabı tüm cihazlarda kolay okunabilir hale getirmek ve kullanıcının yazı tipi boyutunu artırmasına olanak tanımak için, kod listeleme genişliğimizi 47 karakterle sınırlıyoruz. Bu bazen uzlaşmayı gerektirir, ancak sonuçların buna değdiğini düşünüyoruz. Bu genişlikleri elde etmek için, birçok biçimlendirme stilinde yer alabilecek boşlukları kaldırabiliriz—özellikle, standart dört boşluk yerine iki boşluklu girintiler kullanırız.

"Duraklama"

Bazen şunu göreceksiniz:

- -

Bu, bir duraklama veya küçük bir sıfırlama belirtir. Bu kitapta genellikle mevcut alt bölümün kısa bir özetinden önce görünür, ancak "Özet" alt başlığı gereksiz olurdu. Bazı kitaplar bunu, bir fikrin tamamlandığını ve yeni bir şeye başladığımızı belirtmek için kullanır, ancak hala aynı konu içinde ve bir alt bölüm veya yeni bir bölüm açmaya yeterli büyüklükte değil. Leanpub'daki markdown oldukça sınırlıdır ve bir veya daha fazla nokta (orijinal denemem) kullanmak mümkün değildir. Markdown'da iki tire koymak bir nokta ve bir tire üretir. Bunun daha iyi bir yolu olabilir, ama ben bulamadım, bu yüzden buna karar verdim.

Kitaptan Örnek

AtomicKotlin.com adresinde elektronik kitabın ücretsiz bir örneğini sunuyoruz. Örnek, ilk iki bölümü ve birkaç sonraki atomu tam olarak içermektedir. Bu şekilde kitabı deneyebilir ve size uygun olup olmadığına karar verebilirsiniz.

Tam kitap hem basılı kitap olarak hem de eKitap olarak satıştadır. Ücretsiz örnekte yaptıklarımızı beğenirseniz, lütfen kullandığınız şey için ödeme yaparak bizi destekleyin ve çalışmalarımıza devam etmemize yardımcı olun. Kitabın size yardımcı olmasını umar ve desteğinizi takdir ederiz.

İnternet çağında herhangi bir bilgi parçasını kontrol etmek mümkün görünmüyor. Muhtemelen bu kitabın elektronik versiyonunu birçok yerde bulacaksınız. Şu anda

kitabı ödeyemiyorsanız ve bu sitelerden birinden indiriyorsanız, lütfen "iyiliği yaymaya" çalışın. Örneğin, dili öğrendikten sonra başka birinin öğrenmesine yardımcı olun. Ya da herhangi bir şekilde birine yardımcı olun. Belki gelecekte daha iyi durumda olursunuz ve sonra kitabın parasını ödeyebilirsiniz.

Alıştırmalar ve Çözümler

Atomic Kotlin'deki çoğu atom, birkaç küçük alıştırma ile birlikte gelir. Anlayışınızı geliştirmek için, atomu okuduktan hemen sonra alıştırmaları çözmenizi öneririz. Çoğu alıştırma, JetBrains IntelliJ IDEA entegre geliştirme ortamı (IDE) tarafından otomatik olarak kontrol edilir, böylece ilerlemenizi görebilir ve takılırsanız ipuçları alabilirsiniz.

Aşağıdaki bağlantıları http://AtomicKotlin.com/exercises/[4] adresinde bulabilirsiniz.

Alıştırmaları çözmek için, bu eğitimleri takip ederek IntelliJ IDEA'yı ve EduTools eklentisini kurun:

1. IntelliJ IDEA ve EduTools Eklentisini Kurun[5].
2. Atomic Kotlin kursunu açın ve alıştırmaları çözün[6].

Kursta, tüm alıştırmaların çözümlerini bulacaksınız. Bir alıştırmada takılırsanız, ipuçlarını kontrol edin veya çözüme göz atmayı deneyin. Yine de kendiniz uygulamanızı öneririz.

Kursu kurma ve çalıştırma konusunda herhangi bir sorun yaşarsanız, lütfen Sorun Giderme Kılavuzu[7] okuyun. Bu sorununuzu çözmezse, kılavuzda belirtildiği gibi destek ekibiyle iletişime geçin.

Kurs içeriğinde (örneğin, bir görevin testi yanlış sonuç üretiyorsa) bir hata bulursanız, lütfen bu önceden doldurulmuş formu[8] kullanarak sorunu bildirmek için hata takipçimizi kullanın. YouTrack'a giriş yapmanız gerekecektir. Kursu geliştirmemize yardımcı olduğunuz için zamanınızı takdir ediyoruz!

[4] http://AtomicKotlin.com/exercises/
[5] https://www.jetbrains.com/help/education/install-edutools-plugin.html
[6] https://www.jetbrains.com/help/education/learner-start-guide.html?section=Atomic%20Kotlin
[7] https://www.jetbrains.com/help/education/troubleshooting-guide.html
[8] https://youtrack.jetbrains.com/newIssue?project=EDC&summary=AtomicKotlin%3A&c=Subsystem%20Kotlin&c=

Seminerler

Canlı seminerler ve diğer öğrenme araçları hakkında bilgileri *AtomicKotlin.com* adresinde bulabilirsiniz.

Konferanslar

Bruce, Winter Tech Forum[9] gibi *Open-Spaces* konferansları düzenlemektedir. Faaliyetlerimiz ve konuşmalarımız hakkında bilgi almak için *AtomicKotlin.com* adresinde posta listesine katılın.

Bizi Destekleyin

Bu büyük bir projeydi. Bu kitabı ve destekleyici materyalleri üretmek zaman ve çaba gerektirdi. Bu kitabı beğendiyseniz ve benzer şeyler görmek istiyorsanız, lütfen bizi destekleyin:

- **Blog yazın**, **tweet atın vb**. **ve arkadaşlarınıza söyleyin**. Bu, tabandan gelen bir pazarlama çabasıdır, bu yüzden yaptığınız her şey yardımcı olacaktır.
- Bu kitabın **eKitap veya basılı versiyonunu** *AtomicKotlin.com* adresinden satın alın.
- **Diğer destek ürünleri veya etkinlikler** için *AtomicKotlin.com* adresini kontrol edin.

Hakkımızda

Bruce Eckel, çok ödüllü *Thinking in Java* ve *Thinking in C++* kitaplarının ve Atomic Scala[10] gibi diğer birçok bilgisayar programlama kitabının yazarıdır. Dünya çapında yüzlerce sunum yapmış ve Winter Tech Forum[11] gibi alternatif konferanslar

[9] http://www.WinterTechForum.com
[10] http://www.atomicscala.com/
[11] http://www.WinterTechForum.com

ve etkinlikler düzenlemektedir. Bruce, uygulamalı fizik alanında BS ve bilgisayar mühendisliği alanında MS derecelerine sahiptir. Blogu www.BruceEckel.com[12] adresindedir ve danışmanlık, eğitim ve konferans işi Mindview LLC[13] adlı şirkettedir.

Svetlana Isakova, Kotlin derleyici ekibinin bir üyesi olarak başladı ve şimdi JetBrains için geliştirici savunucusu olarak çalışıyor. Kotlin öğretiyor ve dünya çapında konferanslarda konuşmalar yapıyor ve *Kotlin in Action* kitabının ortak yazarıdır.

Teşekkürler

- Kotlin Dil Tasarım Ekibi ve katkıda bulunanlar.
- Bu kitabın yayınlanmasını *çok* daha kolay hale getiren Leanpub geliştiricileri.
- Gradle derlemesini Kotlin'e dönüştüren ve genel anlamda harika olan James Ward.

Adanmışlıklar

Sevgili babam, E. Wayne Eckel için. 1 Nisan 1924—23 Kasım 2016. Bana ilk önce makineler, aletler ve tasarım hakkında bilgi verdin.

Çok erken kaybettiğimiz ve her zaman özleyeceğimiz babam Sergey Lvovich Isakov için.

Kapak Hakkında

Daniel Will-Harris[14], kapak tasarımını Kotlin logosuna dayanarak tasarladı.

[12] http://www.BruceEckel.com
[13] https://www.mindviewllc.com/
[14] http://www.will-harris.com

Neden Kotlin?

> *Programlar, insanların okuyabilmesi için yazılmalı ve sadece makinaların çalıştırması için yazılmamalıdır.*—**Harold Abelson**, *Structure and Interpretation of Computer Programs* kitabının ortak yazarı.

Bu bölüm, Kotlin'in nerede durduğunu ve neden öğrenmek isteyebileceğinizi anlamanız için programlama dillerinin tarihsel gelişiminin genel bir bakışıdır. Eğer acemiyseniz, bu bölümde tanıtılan bazı konular şu anda size çok karmaşık gelebilir. Bu bölümü atlayıp kitabın daha fazla kısmını okuduktan sonra geri dönmekten çekinmeyin.

Programlama dili tasarımı, makinanın ihtiyaçlarına hizmet etmekten programcının ihtiyaçlarına hizmet etmeye doğru evrimsel bir yoldur.

Bir programlama dili, bir dil tasarımcısı tarafından icat edilir ve dili kullanmak için araç olarak görev yapan bir veya daha fazla program olarak uygulanır. Uygulayıcı genellikle başlangıçta dil tasarımcısıdır.

Erken dönem diller donanım sınırlamalarına odaklanmıştır. Bilgisayarlar daha güçlü hale geldikçe, yeni diller güvenilirliğe vurgu yaparak daha sofistike programlamaya doğru kaymaktadır. Bu diller, programlamanın psikolojisine dayalı özellikler seçebilir.

Her programlama dili, bir dizi deneyden oluşur. Tarihsel olarak, programlama dili tasarımı, programcıları daha üretken kılacak şeyler hakkında yapılan bir dizi tahmin ve varsayımdan oluşmuştur. Bu deneylerin bazıları başarısız olur, bazıları hafif bir başarı sağlar ve bazıları çok başarılı olur.

Her yeni dilden deneyimler öğreniriz. Bazı diller, tesadüfi olan konuları ele alır, ya da çevre değişir (daha hızlı işlemciler, daha ucuz bellek, programlama ve diller hakkında yeni anlayışlar) ve bu konu daha az önemli veya hatta önemsiz hale gelir. Eğer bu fikirler eskirse ve dil evrim geçirmezse, kullanımdan düşer.

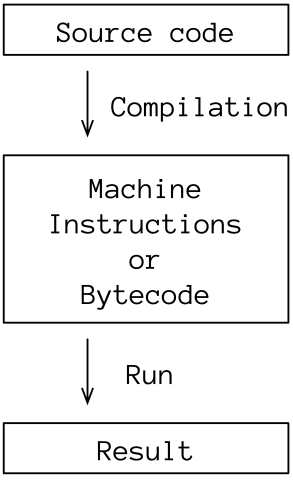
İlk programcılar, işlemci makine talimatlarını temsil eden sayılarla doğrudan çalıştılar. Bu yaklaşım, çok sayıda hata üretti ve sayıları, programcıların daha kolay

hatırlayıp okuyabileceği hafızalı *opcode*'lar—kelimeler ve diğer yardımcı araçlarla değiştiren *makine dili* oluşturuldu. Ancak, hala makine dili talimatları ile makine talimatları arasında birebir bir ilişki vardı ve programcılar her bir satır makine kodunu yazmak zorundaydı. Ayrıca, her bilgisayar işlemcisi kendi özel makine dilini kullanırdı.

Makine dilinde program geliştirmek son derece pahalıdır. Daha yüksek seviyeli diller, düşük seviyeli makine dillerinden uzaklaşarak bir soyutlama seviyesi oluşturarak bu sorunu çözmeye yardımcı olur.

Derleyiciler ve Yorumlayıcılar

Bir yorumlayıcı dilin talimatları, *yorumlayıcı* adı verilen bir program tarafından doğrudan yürütülür. Kotlin, *yorumlanan* değil, *derlenen* bir dildir. Derlenen bir dilin kaynak kodu, kendi başına bir program olarak çalışan veya doğrudan bir donanım işlemcisinde ya da bir işlemciyi taklit eden bir *sanal makine* üzerinde çalışan farklı bir temsile dönüştürülür:



C, C++, Go ve Rust gibi diller, doğrudan altta yatan donanım *merkezi işlem birimi* (CPU) üzerinde çalışan *makine kodu*na derlenir. Java ve Kotlin gibi diller ise, donanım CPU'sunda doğrudan çalışmayan, ancak bir *sanal makine* üzerinde çalışan bir ara seviye format olan *bayt kodu*na derlenir, bu bayt kodu talimatlarını yürüten

bir programdır. Kotlin'in JVM versiyonu tarafından üretilen programlar, *Java Sanal Makinesi* (JVM) üzerinde çalışır.

Taşınabilirlik, sanal bir makinenin önemli bir avantajıdır. Aynı bayt kodu, sanal makineye sahip her bilgisayarda çalışabilir. Sanal makineler, belirli donanımlar için optimize edilebilir ve hız sorunlarını çözebilir. JVM, yılların optimizasyonlarını içerir ve birçok platformda uygulanmıştır.

Derleme zamanında, kod, derleyici tarafından *derleme zamanı hatalarını* keşfetmek için kontrol edilir. (IntelliJ IDEA ve diğer geliştirme ortamları, bu hataları kodu girdiğinizde vurgular, böylece sorunları hızlı bir şekilde keşfedip düzeltebilirsiniz). Eğer derleme zamanı hatası yoksa, kaynak kod bayt koduna derlenir.

Bir *çalışma zamanı hatası*, derleme zamanında tespit edilemez, bu yüzden programı çalıştırdığınızda ortaya çıkar. Genellikle, çalışma zamanı hatalarını keşfetmek daha zordur ve düzeltmek daha pahalıdır. Kotlin gibi *statik tipli diller* mümkün olduğunca çok hatayı derleme zamanında keşfederken, *dinamik diller* güvenlik kontrollerini çalışma zamanında gerçekleştirir (bazı dinamik diller, yapabilecekleri kadar çok güvenlik kontrolü yapmazlar).

Kotlin'i Etkileyen Diller

Kotlin, birçok dilden fikir ve özellikler alır, ve bu diller de önceki dillerden etkilenmiştir. Kotlin'e nasıl geldiğimizi anlamak için bazı programlama dili tarihini bilmek faydalıdır. Burada tanımlanan diller, kendilerinden sonra gelen dillere olan etkileri için seçilmiştir. Bu dillerin tamamı, bazen ne *yapılmaması* gerektiğine dair bir örnek olarak Kotlin'in tasarımına ilham vermiştir.

FORTRAN: FORmula TRANslation (1957)

Bilim insanları ve mühendisler tarafından kullanılması için tasarlanan Fortran'ın amacı, denklemleri kodlamayı kolaylaştırmaktı. İnce ayarlanmış ve test edilmiş Fortran kütüphaneleri bugün hala kullanılmaktadır, ancak genellikle diğer dillerden çağrılabilir hale getirilmek için "sarılır".

LISP: LISt Processor (1958)

LISP, uygulamaya özgü olmaktan ziyade, temel programlama kavramlarını barındırıyordu; bilgisayar bilimcilerinin diliydi ve ilk *fonksiyonel* programlama diliydi (Bu kitapta fonksiyonel programlamayı öğreneceksiniz). Gücü ve esnekliği karşılığında verimlilikten ödün verilmişti: LISP genellikle erken dönem makinelerde çalıştırmak için çok pahalıydı ve yalnızca son yıllarda makineler yeterince hızlı hale geldiğinde LISP'in kullanımı yeniden canlandı. Örneğin, GNU Emacs editörü tamamen LISP ile yazılmıştır ve LISP kullanılarak genişletilebilir.

ALGOL: ALGOrithmic Language (1958)

1950'lerin dilleri arasında tartışmasız en etkili olanı çünkü birçok sonraki dilde kalıcı olan sözdizimini tanıttı. Örneğin, C ve türevleri "ALGOL-benzeri" dillerdir.

COBOL: COmmon Business-Oriented Language (1959)

İş, finans ve idari veri işleme için tasarlanmıştır. İngilizce benzeri bir sözdizimine sahiptir ve kendi kendini belgeleyen ve oldukça okunabilir olması amaçlanmıştır. Bu amaç genellikle başarısız olmuştur—COBOL, yanlış yerleştirilmiş bir nokta nedeniyle ünlü hatalarla bilinir—ABD Savunma Bakanlığı ana bilgisayarlarda yaygın kullanımı zorlamış ve sistemler bugün hala çalışmakta (ve bakım gerektirmektedir).

BASIC: Beginners' All-purpose Symbolic Instruction Code (1964)

BASIC, programlamayı erişilebilir kılma girişimlerinden biriydi. Çok başarılı olmasına rağmen, özellikleri ve sözdizimi sınırlıydı, bu nedenle daha sofistike dilleri öğrenmesi gereken insanlar için yalnızca kısmen yardımcı oldu. Ağırlıklı olarak yorumlanan bir dildir, bu da çalıştırmak için programın orijinal koduna ihtiyaç duyduğunuz anlamına gelir. Buna rağmen, birçok kullanışlı program BASIC ile yazılmıştır, özellikle Microsoft'un "Office" ürünleri için bir betik dili olarak. BASIC, insanların birçok varyasyonunu yaptığı ilk "açık" programlama dili olarak bile düşünülebilir.

Simula 67, the Original Object-Oriented Language (1967)

Bir *simülasyon* tipik olarak birçok "nesne"nin birbirleriyle etkileşimini içerir. Farklı nesneler farklı özelliklere ve davranışlara sahiptir. O dönemde var olan diller, simülasyonlar için kullanılması zor olduğundan, simülasyon nesneleri oluşturmak için doğrudan destek sağlamak üzere Simula (başka bir "ALGOL-benzeri" dil) geliştirildi. Bu fikirlerin genel amaçlı programlama için de yararlı olduğu ortaya çıktı ve bu, Nesne Yönelimli (OO) dillerin doğuşuydu.

Pascal (1970)

Pascal, dili *tek geçişli bir derleyici* olarak uygulanabilir hale getirerek derleme hızını artırdı. Dil programcının kodunu belirli bir şekilde yapılandırmasını zorunlu kıldı ve program organizasyonunda biraz hantal ve daha az okunabilir kısıtlamalar getirdi. İşlemciler hızlandıkça, bellek ucuzladıkça ve derleyici teknolojisi daha iyi hale geldikçe, bu kısıtlamaların etkisi çok maliyetli hale geldi.

Pascal'ın bir uygulaması olan Borland'ın Turbo Pascal'ı, başlangıçta CP/M makinelerinde çalıştı ve ardından erken MS-DOS'a (Windows'un öncüsü) geçiş yaptı, daha sonra Windows için Delphi diline evrildi. Her şeyi bellekte tutarak, Turbo Pascal çok düşük güçlü makinelerde bile çok hızlı derlendi, programlama deneyimini dramatik bir şekilde iyileştirdi. Yaratıcısı Anders Hejlsberg, daha sonra C# ve TypeScript'i tasarlamaya devam etti.

Pascal'ın mucidi Niklaus Wirth, daha sonraki dilleri yarattı: Modula, Modula-2 ve Oberon. Adından da anlaşılacağı gibi, Modula daha iyi organizasyon ve daha hızlı derleme için programları modüllere ayırmaya odaklandı. Çoğu modern dil *ayrı derleme* ve bir tür modül sistemini destekler.

C (1972)

Daha fazla üst düzey dilin artmasına rağmen, programcılar hala montaj dili yazıyordu. Bu genellikle *sistem programlama* olarak adlandırılır, çünkü işletim sistemi seviyesinde yapılır, ancak aynı zamanda özel fiziksel cihazlar için gömülü programlamayı da içerir. Bu sadece zahmetli ve pahalı olmakla kalmaz (Bruce kariyerine gömülü

sistemler için montaj dili yazarak başladı), aynı zamanda taşınabilir değildir—montaj dili yalnızca yazıldığı işlemcide çalışabilir. C, "yüksek seviyeli montaj dili" olarak tasarlandı ve donanıma yeterince yakın olduğu için nadiren montaj dili yazmanız gerekir. Daha da önemlisi, bir C programı, C derleyicisi olan herhangi bir işlemcide çalışır. C, programı işlemciden ayırarak büyük ve pahalı bir sorunu çözdü. Sonuç olarak, eski montaj dili programcıları C'de çok daha verimli hale geldi. C o kadar etkili oldu ki, yeni diller (özellikle Go ve Rust) hala sistem programlama için onu devralmaya çalışıyor.

Smalltalk (1972)

Başlangıçtan itibaren tamamen nesne yönelimli olacak şekilde tasarlanan Smalltalk, deney ve hızlı uygulama geliştirme için bir platform olarak OO (nesne yönelimli) ve dil teorisini önemli ölçüde ileri taşıdı. Ancak, dillerin hala tescilli olduğu bir dönemde oluşturulmuştu ve bir Smalltalk sisteminin giriş fiyatı binlerce dolar olabilirdi. Yorumlanmıştı, bu yüzden programları çalıştırmak için bir Smalltalk ortamına ihtiyacınız vardı. Açık kaynak Smalltalk uygulamaları, programlama dünyası ilerledikten sonra ortaya çıktı. Smalltalk programcıları, C++ ve Java gibi sonraki OO dillerine fayda sağlayan büyük içgörüler sağlamışlardır.

C++: Nesnelerle Daha İyi Bir C (1983)

Bjarne Stroustrup, daha iyi bir C ve Simula-67 kullanırken deneyimlediği nesne yönelimli yapıları destekleyen bir dil istediği için C++'ı yarattı. Bruce, ilk sekiz yıl boyunca C++ Standartları Komitesinin bir üyesiydi ve *Thinking in C++* dahil olmak üzere C++ üzerine üç kitap yazdı.

C++ tasarımının temel prensibi, C ile geriye dönük uyumluluktu; bu nedenle, C kodu neredeyse hiç değişiklik yapılmadan C++'ta derlenebilir. Bu, kolay bir geçiş yolu sağladı — programcılar C ile programlamaya devam edebilir, C++'ın faydalarını elde edebilir ve üretken kalırken C++ özelliklerini yavaş yavaş deneyebilirlerdi. C++'ın çoğu eleştirisi, C ile geriye dönük uyumluluk kısıtlamasına dayanabilir.

C ile ilgili sorunlardan biri *bellek yönetimi* meselesiydi. Programcı önce belleği edinmeli, sonra bu belleği kullanarak bir işlem yapmalı ve ardından belleği serbest bırakmalıydı. Belleği serbest bırakmayı unutmak *bellek sızıntısı* olarak adlandırılır

ve mevcut belleğin tükenmesine ve işlemin çökmesine neden olabilir. C++'ın ilk versiyonu bu alanda bazı yenilikler yaptı ve *kurucular* ile uygun başlatmayı sağladı. Dilin sonraki sürümleri, bellek yönetimi konusunda önemli iyileştirmeler yapmıştır.

Python: Dost Canlısı ve Esnek (1990)

Python'un tasarımcısı Guido Van Rossum, dili "herkes için programlama" ilhamıyla oluşturdu. Python topluluğunu beslemesi, onu programlama dünyasında en dost canlısı ve destekleyici topluluk olarak tanınır hale getirdi. Python, hemen hemen her platformda, gömülü sistemler ve makine öğrenimi dahil olmak üzere uygulamalarla sonuçlanan ilk açık kaynak dillerden biriydi. Dinamikliği ve kullanımı kolay olması, onu küçük, tekrarlayan görevleri otomatikleştirmek için ideal kılar, ancak özellikleri aynı zamanda büyük ve karmaşık programların oluşturulmasını da destekler.

Python gerçek bir "taban dili"dir; hiçbir zaman bir şirket tarafından tanıtılmadı ve hayranlarının tavrı dili asla zorlamamak, sadece öğrenmek isteyen herkese yardım etmekti. Dil sürekli olarak gelişmeye devam ediyor ve son yıllarda popülaritesi hızla arttı.

Python, işlevsel ve OO programlamayı birleştiren ilk ana akım dil olabilir. Otomatik bellek yönetimi için *çöp toplama* kullanarak Java'dan önce gelmiştir (genellikle belleği kendiniz ayırmanız veya serbest bırakmanız gerekmez) ve programları birden fazla platformda çalıştırma yeteneğine sahiptir.

Haskell: Saf Fonksiyonel Programlama (1990)

Miranda (1985) adlı tescilli dilden ilham alan Haskell, saf fonksiyonel programlama araştırmaları için açık bir standart olarak oluşturuldu, ancak ürünler için de kullanılmıştır. Haskell'den alınan sözdizimi ve fikirler, Kotlin dahil olmak üzere çok sayıda sonraki dili etkilemiştir.

Java: Sanal Makineler ve Çöp Toplama (1995)

James Gosling ve ekibi, bir TV set üstü kutusu için kod yazma görevi verildi. C++'ı sevmediklerine karar verdiler ve kutuyu yapmak yerine Java dilini oluşturdular. Sun Microsystems şirketi, yeni bir fikir olan ücretsiz dilin (halen yeni bir fikir olan) ortaya

çıkan İnternet manzarasını hakimiyet altına almak için büyük bir pazarlama itici gücü koydu.

Bu algılanan İnternet hakimiyeti zaman penceresi, Java dil tasarımı üzerinde çok fazla baskı oluşturdu ve bu da önemli sayıda kusurla sonuçlandı (Kitap *Thinking in Java*, okuyucuların bu kusurlarla baş etmeye hazırlıklı olmalarını sağlamak için bu kusurları aydınlatır). Oracle'da Java'nın şu anki baş geliştiricisi olan Brian Goetz, devraldığı kısıtlamalara rağmen Java'da dikkate değer ve şaşırtıcı iyileştirmeler yapmıştır. Java son derece başarılı olmasına rağmen, önemli bir Kotlin tasarım hedefi, Java'nın kusurlarını düzeltmek ve programcıların daha üretken olmasını sağlamaktır.

Java'nın başarısı iki yenilikçi özelliğinden geldi: bir *sanal makine* ve *çöp toplama*. Bu özellikler diğer dillerde mevcuttu—örneğin, LISP, Smalltalk ve Python çöp toplama işlemi yapıyor ve UCSD Pascal bir sanal makine üzerinde çalışıyordu—ancak ana akım diller için pratik olarak görülmüyordu. Java bunu değiştirdi ve bunu yaparken programcıları önemli ölçüde daha üretken hale getirdi.

Bir sanal makine, dil ile donanım arasında bir ara katmandır. Dil, belirli bir işlemci için makine kodu üretmek zorunda değildir; yalnızca sanal makine üzerinde çalışan bir ara dili (baytkodu) üretmesi yeterlidir. Sanal makineler işlem gücü gerektirir ve Java'dan önce pratik olmadığına inanılıyordu. *Java Sanal Makinesi* (JVM), Java'nın sloganı "bir kez yaz, her yerde çalıştır" sözüne yol açtı. Ek olarak, diğer diller JVM'yi hedefleyerek daha kolay geliştirilebilir; örneğin, Java benzeri bir betik dili olan Groovy ve bir LISP sürümü olan Clojure.

Çöp toplayıcı, belleği serbest bırakmayı unutma sorununu çözer veya bir depolama alanının ne zaman kullanılmadığını bilmenin zor olduğu durumları yönetir. Bellek sızıntıları nedeniyle projeler önemli ölçüde gecikmiş veya hatta iptal edilmiştir. Çöp toplayıcı, bazı önceki dillerde görünmesine rağmen, Java, onun uygulanabilirliğini gösterene kadar kabul edilemez bir miktarda yük oluşturduğu düşünülüyordu.

JavaScript: Sadece Adıyla Java (1995)

İlk Web tarayıcısı, bir Web sunucusundan sayfaları kopyalayıp görüntülerdi. Web tarayıcıları çoğaldı ve dil desteği gerektiren yeni bir programlama platformu haline geldi. Java bu dil olmak istedi ama iş için çok hantal kaldı. JavaScript, LiveScript olarak başladı ve ilk Web tarayıcılarından biri olan NetScape Navigator'a entegre

edildi. Adının JavaScript olarak değiştirilmesi, NetScape'in pazarlama taktiğiydi, çünkü dilin Java ile yalnızca belirsiz bir benzerliği vardı.

Web yaygınlaştıkça, JavaScript olağanüstü derecede önemli hale geldi. Ancak, JavaScript'in davranışı o kadar öngörülemezdi ki Douglas Crockford, dildeki tüm sorunları göstererek programcıların bunlardan kaçınmalarını sağlamak için alaycı bir başlık olan *JavaScript, İyi Kısımlar* adlı bir kitap yazdı. ECMAScript komitesinin sonraki iyileştirmeleri, JavaScript'i orijinal bir JavaScript programcısına tanınmaz hale getirdi. Şimdi stabil ve olgun bir dil olarak kabul edilir.

Web assembly (WASM), web tarayıcıları için bir tür byte kodu olarak JavaScript'ten türetilmiştir. Genellikle JavaScript'ten çok daha hızlı çalışır ve diğer diller tarafından üretilebilir. Bu yazı yazılırken, Kotlin ekibi WASM'ı bir hedef olarak eklemek için çalışıyor.

C#: .NET İçin Java (2000)

C#, .NET (Windows) platformunda Java'nın bazı önemli yeteneklerini sağlamak için tasarlandı, ancak tasarımcıları Java dilini takip etme kısıtlamasından kurtardı. Sonuç, Java üzerinde sayısız iyileştirme içeriyordu. Örneğin, C#, Kotlin'de yoğun olarak kullanılan *uzantı fonksiyonları* kavramını geliştirdi. C# ayrıca Java'dan çok daha işlevsel hale geldi. Birçok C# özelliği açıkça Kotlin tasarımını etkiledi.

Scala: SCALAble (2003)

Martin Odersky, Scala'yı Java sanal makinesinde çalıştırmak için yarattı: JVM üzerinde yapılan çalışmalardan yararlanmak, Java programlarıyla etkileşimde bulunmak ve belki de Java'yı yerinden etmek fikriyle. Bir araştırmacı olarak, Odersky ve ekibi, özellikle Java'ya dahil edilmeyen dil özellikleriyle denemeler yapmak için Scala'yı bir platform olarak kullandılar.

Bu deneyler aydınlatıcıydı ve bunların birçoğu genellikle değiştirilmiş bir formda Kotlin'e dahil edildi. Örneğin, özel durumlarda kullanılmak üzere + gibi operatörleri yeniden tanımlama yeteneği, *operatör aşırı yükleme* olarak adlandırılır. Bu, C++'da dahil edildi ancak Java'da değil. Scala, operatör aşırı yüklemesini ekledi, ancak ayrıca herhangi bir karakter dizisiyle yeni operatörler icat etmenize izin verir. Bu genellikle

kafa karıştırıcı kodlar üretir. Kotlin'e sınırlı bir operatör aşırı yükleme formu dahil edildi, ancak yalnızca mevcut operatörleri aşırı yükleyebilirsiniz.

Scala ayrıca Python gibi bir nesne-fonksiyonel hibrittir, ancak saf fonksiyonlar ve katı nesnelere odaklanır. Bu, Kotlin'in de bir nesne-fonksiyonel hibrit olma tercihini ilham verdi.

Scala gibi, Kotlin JVM üzerinde çalışır ancak Java ile Scala'dan çok daha kolay etkileşime girer (bkz. Ek B). Ayrıca, Kotlin JavaScript'i, Android OS'yi hedefler ve diğer platformlar için yerel kod üretir.

Atomic Kotlin, Atomic Scala[15]'daki fikirler ve materyallerden evrilmiştir.

Groovy: Dinamik bir JVM Dili (2007)

Dinamik diller caziptir çünkü statik dillere göre daha etkileşimli ve özdürler. JVM üzerinde daha dinamik bir programlama deneyimi üretmek için birçok girişim olmuştur, bunlara Jython (Python) ve Clojure (bir Lisp lehçesi) dahildir. Groovy, geniş kabul gören ilk dil oldu.

İlk bakışta, Groovy, Java'nın temizlenmiş bir versiyonu gibi görünür ve daha hoş bir programlama deneyimi sunar. Çoğu Java kodu, Groovy'de değişmeden çalışır, bu nedenle Java programcıları hızlı bir şekilde verimli olabilir ve daha sonra Java'ya göre önemli programlama iyileştirmeleri sağlayan daha sofistike özellikleri öğrenebilirler.

Boşluk problemine çözüm getiren Kotlin operatörleri `?.` ve `?:` ilk olarak Groovy'de ortaya çıktı.

Kotlin'de tanınabilir birçok Groovy özelliği vardır. Bu özelliklerin bazıları diğer dillerde de görülür ve muhtemelen Kotlin'e dahil edilmeleri için daha fazla baskı yapmışlardır.

Neden Kotlin? (2011'de Tanıtıldı, Sürüm 1.0: 2016)

C++'ın başlangıçta "daha iyi bir C" olması amaçlandığı gibi, Kotlin de başlangıçta "daha iyi bir Java" olma yönünde eğilimliydi. Bu hedeften çok öteye evrildi.

[15] http://www.AtomicScala.com

Kotlin, diğer programlama dillerinin en başarılı ve yararlı özelliklerini seçip kullanır—bu özellikler sahada test edildikten ve özellikle değerli oldukları kanıtlandıktan sonra.

Bu nedenle, başka bir dilden geliyorsanız, Kotlin'de o dilin bazı özelliklerini tanıyabilirsiniz. Bu kasıtlıdır: Kotlin, test edilmiş kavramları kullanarak verimliliği maksimize eder.

Okunabilirlik

Okunabilirlik, dilin tasarımında birincil hedeftir. Kotlin sözdizimi kısadır—çoğu senaryo için tören gerektirmez, ancak yine de karmaşık fikirleri ifade edebilir.

Araçlar

Kotlin, geliştirici araçları konusunda uzmanlaşmış bir şirket olan JetBrains'ten gelir. İlk sınıf araç desteğine sahiptir ve birçok dil özelliği araçlar düşünülerek tasarlanmıştır.

Çoklu-Paradigma

Kotlin, bu kitapta nazikçe tanıtılan birden fazla programlama paradigmasını destekler:

- Emirsel programlama
- Fonksiyonel programlama
- Nesne yönelimli programlama

Çoklu-Platform

Kotlin kaynak kodu farklı hedef platformlara derlenebilir:

- **JVM**. Kaynak kod JVM bayt koduna (`.class` dosyaları) derlenir ve ardından herhangi bir Java Sanal Makinesi (JVM) üzerinde çalıştırılabilir.
- **Android**. Android'in kendi çalışma zamanı vardır, ART[16] (öncüsü Dalvik olarak adlandırılır). Kotlin kaynak kodu *Dalvik Çalıştırılabilir Biçimi*ne (`.dex` dosyaları) derlenir.

[16] https://source.android.com/devices/tech/dalvik

- **JavaScript**, bir web tarayıcısı içinde çalıştırmak için.
- **Yerel İkili Dosyalar**, belirli platformlar ve CPU'lar için makine kodu üreterek.

Bu kitap, yalnızca JVM'yi hedef platform olarak kullanarak dili kendisini odaklanır. Dili öğrendikten sonra, Kotlin'i farklı uygulama ve hedef platformlara uygulayabilirsiniz.

İki Kotlin Özelliği

Bu atom, programcı olduğunuzu varsaymaz, bu da Kotlin'in alternatiflere göre çoğu faydasını açıklamayı zorlaştırır. Ancak, bu erken aşamada açıklanabilecek iki konu çok etkilidir: Java ile uyumluluk ve "değer yok" göstergesinin sorunu.

Zahmetsiz Java ile Uyumluluk

"C'nin daha iyi bir versiyonu" olmak için, C++'ın C'nin sözdizimi ile geriye dönük uyumlu olması gerekir, ancak Kotlin'in Java'nın sözdizimi ile geriye dönük uyumlu olması gerekmez—sadece JVM ile çalışması gerekir. Bu, Kotlin tasarımcılarının çok daha temiz ve güçlü bir sözdizimi yaratmalarını sağlar, Java'yı karıştıran görsel gürültü ve karmaşıklık olmadan.

Kotlin'in "daha iyi bir Java" olması için, deneme deneyiminin hoş ve sürtünmesiz olması gerekir, bu nedenle Kotlin mevcut Java projeleriyle zahmetsiz entegrasyon sağlar. Küçük bir Kotlin işlevselliği yazabilir ve mevcut Java kodunuzun arasına yerleştirebilirsiniz. Java kodu, Kotlin kodunun orada olduğunu bile bilmez—sadece daha fazla Java kodu gibi görünür.

Şirketler genellikle yeni bir dili araştırmak için o dilde bağımsız bir program oluştururlar. İdeal olarak, bu program faydalıdır ancak zorunlu değildir, bu nedenle proje başarısız olursa en az zararla sonlandırılabilir. Her şirket bu tür bir deneme için gerekli kaynakları harcamak istemez. Kotlin, mevcut bir Java sistemine sorunsuz bir şekilde entegre olduğu için (ve bu sistemin testlerinden faydalandığı için), Kotlin'i denemek çok ucuz veya hatta ücretsiz hale gelir.

Ek olarak, Kotlin'i yaratan şirket JetBrains, hem Java hem de Kotlin'i destekleyen ve ikisini kolayca entegre etme yeteneğine sahip "Topluluk" (ücretsiz) bir sürümde

IntelliJ IDEA sağlar. Hatta Java kodunu alıp (çoğunlukla) Kotlin'e yeniden yazan bir araca sahiptir.

Ek B Java ile uyumluluğu kapsar.

Boşluğu Temsil Etmek

Özellikle yararlı bir Kotlin özelliği, zorlu bir programlama sorununa getirdiği çözümdür.

Birisi size bir sözlük verdiğinde ve var olmayan bir kelimeyi aramanızı istediğinde ne yaparsınız? Bilinmeyen kelimeler için tanımlar uydurarak sonuçları garanti edebilirsiniz. Daha faydalı bir yaklaşım, "Bu kelime için tanım yok" demektir. Bu, programlamada önemli bir sorunu gösterir: Başlatılmamış bir depolama parçası veya bir işlemin sonucu için "değer yok" nasıl belirtilir?

Boş referans 1965 yılında ALGOL için Tony Hoare tarafından icat edildi ve daha sonra buna "milyar dolarlık hatam" dedi. Bir sorun çok basit olmasıydı—bazen bir odanın boş olduğunu söylemek yeterli değildir. Örneğin, *neden* boş olduğunu bilmeniz gerekebilir. Bu, ikinci sorunu getirir: uygulama. Verimlilik adına, genellikle yalnızca küçük bir miktar belleğe sığabilecek özel bir değerdi ve bu bilgi için zaten ayrılmış olan bellekten daha iyisi yoktu.

Orijinal C dili, depolamayı otomatik olarak başlatmazdı, bu da birçok soruna neden oldu. C++, yeni tahsis edilen depolamayı sıfırlarla doldurarak durumu iyileştirdi. Böylece, bir sayısal değer başlatılmamışsa, sadece sayısal bir sıfır olur. Bu, o kadar kötü görünmüyordu ama başlatılmamış değerlerin sessizce gözden kaçmasına izin verdi (yeni C ve C++ derleyicileri genellikle bu konuda sizi uyarır). Daha kötüsü, eğer bir depolama parçası bir *işaretçi* ise—başka bir depolama parçasını göstermek için kullanılır ("işaret eder")—bir null işaretçi bellekte sıfır konumunu işaret ederdi, bu neredeyse kesinlikle istediğiniz şey değildir.

Java, çalışma zamanında bu tür hataları raporlayarak başlatılmamış değerlere erişimleri önler. Bu, başlatılmamış değerleri keşfetse de, sorunu çözmez çünkü programınızın çöküp çökmeyeceğini doğrulamanın tek yolu onu çalıştırmaktır. Java kodunda bu tür hatalardan çokça bulunur ve programcılar onları bulmak için büyük miktarda zaman harcarlar.

Kotlin, program çalışmadan önce *derleme zamanında* null hatalarına neden olabilecek işlemleri engelleyerek bu sorunu çözer. Bu, Java programcılarının Kotlin'i

benimserken en çok takdir ettikleri özelliktir. Bu tek özellik, Java'nın null hatalarını en aza indirebilir veya ortadan kaldırabilir ve projenizin önemli miktarda zaman ve para tasarrufu yapmasını sağlar.

Bolca Faydalar

Burada açıklayabildiğimiz iki özellik (daha fazla programlama bilgisi gerektirmeden), Java programcısı olsanız da olmasanız da büyük bir fark yaratır. Eğer Kotlin sizin ilk dilinizse ve daha fazla programcıya ihtiyaç duyduğunuz bir projeye katılırsanız, var olan birçok Java programcısından birini Kotlin'e dahil etmek çok daha kolaydır.

Kotlin'in, daha fazla programlama bilgisi edinene kadar açıklayamayacağımız birçok başka faydası vardır. Kitabın geri kalanı bunun içindir.

- -

Diller genellikle tutkuyla seçilir, mantıkla değil... Kotlin'i bir sebep için sevilen bir dil yapmaya çalışıyorum.—Andrey Breslav, Kotlin Baş Dil Tasarımcısı.

Merhaba, Dünya!

> "Merhaba, dünya!" programı, programlama dillerinin temel sözdizimini göstermek için yaygın olarak kullanılır.

Bu programı parçalarını anlamanız için birkaç adımda geliştiriyoruz.

Öncelikle, hiçbir şey yapmayan boş bir programı inceleyelim:

```
// HelloWorld/EmptyProgram.kt

fun main() {
  // Program code here ...
}
```

Örnek, Kotlin tarafından yok sayılan ve aydınlatıcı metin olan bir *yorum* ile başlar. `//` (iki eğik çizgi) mevcut satırın sonuna kadar giden bir yorumu başlatır:

```
// Single-line comment
```

Kotlin, `//` işaretini ve ondan sonraki her şeyi satır sonuna kadar görmezden gelir. Bir sonraki satırda, tekrar dikkat eder.

Bu kitapta her örneğin ilk satırı, kaynak kod dosyasını içeren alt dizinin adıyla (`HelloWorld`) başlayan ve ardından dosyanın adı olan bir yorumdur: `EmptyProgram.kt`. Her atom için örnek alt dizin, o atomun adıyla örtüşür.

anahtar kelimeler dil tarafından ayrılmıştır ve özel bir anlam verilmiştir. `fun` anahtar kelimesi *function* kelimesinin kısaltmasıdır. Bir fonksiyon, o fonksiyonun adı kullanılarak çalıştırılabilen bir kod koleksiyonudur (kitap boyunca fonksiyonlar üzerinde çok zaman harcıyoruz). Fonksiyonun adı `fun` anahtar kelimesinden sonra gelir, bu durumda adı `main()`'dir (düz yazıda, fonksiyon adını parantezlerle takip ederiz).

`main()` aslında bir fonksiyon için özel bir addır; bir Kotlin programının "giriş noktası"nı belirtir. Bir Kotlin programı, birçok farklı isme sahip birçok fonksiyona sahip olabilir, ancak programı çalıştırdığınızda otomatik olarak çağrılan `main()`'dir.

parametre listesi fonksiyon adını takip eder ve parantezlerle çevrilidir. Burada, `main()`'e hiçbir şey geçmiyoruz, bu yüzden parametre listesi boştur.

fonksiyon gövdesi parametre listesinden sonra görünür. Bir açılış parantezi ({) ile başlar ve bir kapanış parantezi (}) ile biter. Fonksiyon gövdesi *ifadeler* ve *durumlar* içerir. Bir durum bir etki üretir ve bir ifade bir sonuç verir.

`EmptyProgram.kt` dosyasının gövdesinde hiçbir ifade veya durum yoktur, sadece bir yorum vardır.

Programın "Hello, world!" görüntülemesini sağlamak için `main()` gövdesine bir satır ekleyelim:

```
// HelloWorld/HelloWorld.kt

fun main() {
  println("Hello, world!")
}
/* Output:
Hello, world!
*/
```

Selamlamayı gösteren satır `println()` ile başlar. `main()` gibi, `println()` da bir fonksiyondur. Bu satır, fonksiyonu *çağırır* ve fonksiyonun gövdesi çalışır. Fonksiyon adını verir, ardından parantezler içinde bir veya daha fazla parametre eklenir. Bu kitapta, düzyazıda bir fonksiyona atıfta bulunurken, bir fonksiyon olduğunun hatırlatıcısı olarak adının ardından parantez ekliyoruz. Burada, `println()` diyoruz.

`println()` tek bir parametre alır, bu bir `String`dir. Bir `String` tanımlamak için karakterleri tırnak işaretleri içine alırsınız.

`println()` parametresini görüntüledikten sonra imleci yeni bir satıra taşır, bu nedenle sonraki çıktı bir sonraki satırda görünür. Bunun yerine, imleci aynı satırda bırakan `print()` kullanabilirsiniz.

Bazı dillerin aksine, Kotlin'de bir ifadenin sonunda noktalı virgül gerekmez. Aynı satıra birden fazla ifade koymadığınız sürece (bu önerilmez) gerekli değildir.

Kitaptaki bazı örneklerde, çıktıyı listenin sonunda, *çok satırlı yorum* içinde gösteriyoruz. Çok satırlı bir yorum, `/*` (bir eğik çizgi ve ardından bir yıldız işareti) ile başlar ve bir `*/` (bir yıldız işareti ve ardından bir eğik çizgi) ile yorum bitene kadar—satır sonları dahil (biz bunlara *yeni satırlar* diyoruz)—devam eder:

```
/* A multiline comment
Doesn't care
about newlines */
```

Kapanış `*/` açıklamasının *ardından* aynı satıra kod eklemek mümkündür, ancak bu kafa karıştırıcıdır, bu yüzden insanlar genellikle bunu yapmazlar.

Açıklamalar, kodu okumaktan açıkça anlaşılmayan bilgileri ekler. Eğer açıklamalar sadece kodun söylediklerini tekrar ediyorsa, rahatsız edici hale gelirler ve insanlar onları göz ardı etmeye başlar. Kod değiştiğinde, programcılar genellikle açıklamaları güncellemeyi unutur, bu yüzden açıklamaları dikkatli kullanmak iyi bir uygulamadır, özellikle kodunuzun karmaşık yönlerini vurgulamak için.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

`var` & `val`

> Bir tanımlayıcı veri tuttuğunda, yeniden atanıp atanamayacağına karar vermeniz gerekir.

Programınızda öğelere başvurmak için *tanımlayıcılar* oluşturursunuz. Bir veri tanımlayıcısı için en temel karar, içeriklerinin program yürütülmesi sırasında değişip değişemeyeceği veya sadece bir kez atanabileceğidir. Bu, iki anahtar kelime ile kontrol edilir:

- `var`, *değişken* kelimesinin kısaltmasıdır ve içeriğinin yeniden atanabileceği anlamına gelir.
- `val`, *değer* kelimesinin kısaltmasıdır ve sadece başlatılabileceği, yeniden atanamayacağı anlamına gelir.

Bir `var` şu şekilde tanımlanır:

```
var identifier = initialization
```

`var` anahtar kelimesi, bir tanımlayıcı, bir eşittir işareti ve ardından başlangıç değeri ile takip edilir. Tanımlayıcı bir harf veya alt çizgi ile başlar, ardından harfler, sayılar ve alt çizgiler gelir. Büyük ve küçük harfler ayırt edilir (yani `thisvalue` ve `thisValue` farklıdır).

İşte bazı `var` tanımlamaları:

```
// VarAndVal/Vars.kt

fun main() {
  var whole = 11              // [1]
  var fractional = 1.4        // [2]
  var words = "Twas Brillig"  // [3]
  println(whole)
  println(fractional)
  println(words)
}
/* Output:
11
1.4
Twas Brillig
*/
```

Bu kitapta satırları köşeli parantezler içinde yorumlanmış numaralarla işaretliyoruz, böylece metinde bu şekilde onlara atıfta bulunabiliriz:

- **[1]** `whole` adında bir `var` oluştur ve içine `11` sakla.
- **[2]** "kesirli sayı" `1.4`'ü `fractional` adlı `var`'a sakla.
- **[3]** Bir metni (bir `String`) `words` adlı `var`'a sakla.

`println()`'ın tek bir değeri argüman olarak alabileceğini unutmayın.

Değişken isminin de ima ettiği gibi, bir `var` değişebilir. Yani, bir `var`'da saklanan veriyi değiştirebilirsiniz. Bir `var`'ın *değişken* olduğunu söyleriz:

```
// VarAndVal/AVarIsMutable.kt

fun main() {
  var sum = 1
  sum = sum + 2
  sum += 3
  println(sum)
}
/* Output:
6
*/
```

`sum = sum + 2` ataması, `sum`'un mevcut değerini alır, iki ekler ve sonucu tekrar `sum` içine atar.

`sum += 3` ataması, `sum = sum + 3` ile aynı anlama gelir. `+=` operatörü, `sum` içinde saklanan önceki değeri alır ve onu `3` artırır, sonra bu yeni sonucu tekrar `sum` içine atar.

Bir `var` içinde saklanan değeri değiştirmek, değişiklikleri ifade etmenin kullanışlı bir yoludur. Ancak, bir programın karmaşıklığı arttığında, tanımlayıcılarınız tarafından temsil edilen değerler değişemezse—yani yeniden atama yapılamazsa—kodunuz daha net, daha güvenli ve anlaşılması daha kolay olur. Değişmeyen bir tanımlayıcıyı `var` yerine `val` anahtar kelimesi kullanarak belirtiriz. Bir `val` yalnızca oluşturulduğunda bir kez atanabilir:

```
val identifier = initialization
```

`val` anahtar kelimesi *değer* kelimesinden gelir ve değiştirilemeyecek bir şeyi ifade eder—*değiştirilemez*dir. Mümkün olduğunca `val` yerine `var` kullanmayı tercih edin. Bu atomun başındaki `Vars.kt` örneği `val`ler kullanılarak yeniden yazılabilir:

```
// VarAndVal/Vals.kt

fun main() {
  val whole = 11
  // whole = 15 // Error   // [1]
  val fractional = 1.4
  val words = "Twas Brillig"
  println(whole)
  println(fractional)
  println(words)
}
/* Output:
11
1.4
Twas Brillig
*/
```

- **[1]** Bir `val` başlattığınızda, yeniden atama yapamazsınız. `whole`'u farklı bir sayıya yeniden atamaya çalışırsak, Kotlin "Val yeniden atanamaz" diyerek şikayet eder.

Tanımlayıcılarınız için açıklayıcı isimler seçmek, kodunuzu daha anlaşılır hale getirir ve genellikle yorumlara olan ihtiyacı azaltır. `Vals.kt` dosyasında, `whole`'un neyi

temsil ettiğine dair hiçbir fikriniz yok. Programınız 11 sayısını kahve içtiğiniz zamanı temsil etmek için saklıyorsa, bunu `coffeetime` olarak adlandırmak başkaları için daha açıktır ve Kotlin tarzını takip ederek ilk harfi küçük yaparsak `coffeeTime` olarak okumak daha kolaydır.

- -

Program çalışırken verilerin değişmesi gerektiğinde `var`lar kullanışlıdır. Bu yaygın bir gereksinim gibi görünüyor, ancak pratikte kaçınılabilir olduğu ortaya çıkıyor. Genel olarak, programlarınızı genişletmek ve bakımını yapmak daha kolaydır eğer `val`ları kullanırsanız. Ancak, nadir durumlarda sadece `val`lar kullanarak bir sorunu çözmek çok karmaşık olabilir. Bu nedenle, Kotlin size `var`ların esnekliğini sunar. Ancak, `val`larla daha fazla zaman geçirdikçe, neredeyse hiç `var`lara ihtiyaç duymadığınızı ve programlarınızın onlarsız daha güvenli ve daha güvenilir olduğunu keşfedeceksiniz.

Egzersizler ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Veri Tipleri

Veriler farklı *tiplerde* olabilir.

Bir matematik problemini çözmek için, bir ifade yazarsınız:

```
5.9 + 6
```

Bu sayıları toplamanın başka bir sayı ürettiğini biliyorsunuz. Kotlin de bunu biliyor. Birinin kesirli bir sayı (`5.9`) olduğunu, Kotlin'in bunu `Ondalık` olarak adlandırdığını ve diğerinin tam sayı (`6`) olduğunu, Kotlin'in bunu `Tamsayı` olarak adlandırdığını biliyorsunuz. Sonucun kesirli bir sayı olduğunu biliyorsunuz.

Bir *tip* (aynı zamanda *veri tipi* olarak da adlandırılır), Kotlin'e bu veriyi nasıl kullanmayı düşündüğünüzü söyler. Bir tip, o tipteki bir ifadenin üretebileceği değerler kümesini tanımlar. Bir tip ayrıca veri üzerinde yapılabilecek işlemleri, verinin anlamını ve o tipe ait değerlerin nasıl saklanabileceğini tanımlar.

Kotlin, ifadelerinizin doğru olduğundan emin olmak için tipleri kullanır. Yukarıdaki ifadede, Kotlin sonucu tutmak için `Ondalık` tipinde yeni bir değer oluşturur.

Kotlin, ihtiyaçlarınıza uyum sağlamaya çalışır. Tip kurallarını ihlal eden bir şey yapmasını isterseniz, bir hata mesajı üretir. Örneğin, bir `String` ve bir sayıyı toplama denemesi yapın:

```
// DataTypes/StringPlusNumber.kt

fun main() {
  println("Sally" + 5.9)
}
/* Output:
Sally5.9
*/
```

Tipler Kotlin'e onları nasıl doğru kullanacağını söyler. Bu durumda, tip kuralları Kotlin'e bir sayıyı bir `String` ile nasıl toplayacağını belirtir: iki değeri ekleyerek ve sonucu tutacak bir `String` oluşturarak.

Şimdi `String` ve `Double` çarpmayı deneyin; `StringPlusNumber.kt` dosyasındaki `+` işaretini `*` ile değiştirin:

```
"Sally" * 5.9
```

Kotlin için bu şekilde türleri birleştirmek mantıklı değildir, bu yüzden bir hata verir.

`var` ve `val` içinde, birkaç tür sakladık. Kotlin, onları nasıl kullandığımıza göre türleri bizim için belirledi. Bu, *tür çıkarımı* olarak adlandırılır.

Daha detaylı olabilir ve türü belirtebiliriz:

```
val identifier: Type = initialization
```

`val` veya `var` anahtar kelimesiyle başlarsınız, ardından tanımlayıcı, iki nokta üst üste, tür, bir `=` ve başlatma değeri gelir. Bu yüzden şöyle demek yerine:

```
val n = 1
var p = 1.2
```

Şunu diyebilirsin:

```
val n: Int = 1
var p: Double = 1.2
```

Kotlin'e `n`'nin bir `Int` ve `p`'nin bir `Double` olduğunu, türünü çıkarmasına izin vermek yerine söyledik.

İşte Kotlin'un temel türlerinden bazıları:

```
// DataTypes/Types.kt

fun main() {
  val whole: Int = 11              // [1]
  val fractional: Double = 1.4     // [2]
  val trueOrFalse: Boolean = true  // [3]
  val words: String = "A value"    // [4]
  val character: Char = 'z'        // [5]
  val lines: String = """Triple quotes let
you have many lines
in your string"""                  // [6]
  println(whole)
```

```
  println(fractional)
  println(trueOrFalse)
  println(words)
  println(character)
  println(lines)
}
/* Output:
11
1.4
true
A value
z
Triple quotes let
you have many lines
in your string
*/
```

- **[1]** `Int` veri tipi bir *tam sayı*dır, yani yalnızca tam sayıları tutar.
- **[2]** Kesirli sayılar tutmak için `Double` kullanın.
- **[3]** `Boolean` veri tipi yalnızca iki özel değer olan `true` ve `false` değerlerini tutar.
- **[4]** `String` bir karakter dizisini tutar. Bir değeri, çift tırnak içine alınmış bir `String` ile atarsınız.
- **[5]** `Char` bir karakter tutar.
- **[6]** Eğer birden fazla satır ve/veya özel karakterleriniz varsa, bunları üçlü çift tırnak içine alın (bu *üçlü tırnaklı bir `String`*tir).

Kotlin, karışık türlerin anlamını belirlemek için tür çıkarımı kullanır. Örneğin, toplama sırasında `Int` ve `Double` karışımı kullanıldığında, Kotlin ortaya çıkan değerin türünü belirler:

```
// DataTypes/Inference.kt

fun main() {
  val n = 1 + 1.2
  println(n)
}
/* Output:
2.2
*/
```

Bir `Int` ve bir `Double`ı tip çıkarımı kullanarak topladığınızda, Kotlin sonucun `Double` olduğunu belirler ve `Double` kurallarına uyduğundan emin olur.

Kotlin'in tip çıkarımı, programcı için iş yapma stratejisinin bir parçasıdır. Tip bildirimi yapmazsanız, Kotlin genellikle bunu çıkarabilir.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Fonksiyonlar

> Bir *fonksiyon*, kendi ismine sahip küçük bir program gibidir ve bu isim başka bir fonksiyondan çağrılarak çalıştırılabilir (*çağrılabilir*).

Bir fonksiyon, bir grup etkinliği birleştirir ve programlarınızı organize etmenin ve kodu yeniden kullanmanın en temel yoludur.

Bir fonksiyona bilgi aktarırsınız ve fonksiyon bu bilgiyi kullanarak hesaplama yapar ve bir sonuç üretir. Bir fonksiyonun temel formu şöyledir:

```
fun functionName(p1: Type1, p2: Type2, ...): ReturnType {
  lines of code
  return result
}
```

`p1` ve `p2` *parametreler*dir: fonksiyona aktardığınız bilgilerdir. Her parametrenin bir tanımlayıcı adı vardır (`p1`, `p2`) ve ardından iki nokta üst üste ve bu parametrenin türü gelir. Parametre listesinin kapanış parantezi, bir iki nokta üst üste ve fonksiyonun ürettiği sonucun türü ile takip edilir. *Fonksiyon gövdesi*ndeki kod satırları süslü parantezler içinde yer alır. `return` anahtar kelimesinin ardından gelen ifade, fonksiyon tamamlandığında üretilen sonuçtur.

Bir parametre, bir fonksiyona neyin aktarıldığını tanımlama şeklinizdir—bu, yer tutucudur. Bir argüman, fonksiyona aktardığınız gerçek değerdir.

Ad, parametreler ve geri dönüş türünün kombinasyonuna *fonksiyon imzası* denir.

İşte `multiplyByTwo()` adlı basit bir fonksiyon:

```
// Functions/MultiplyByTwo.kt

fun multiplyByTwo(x: Int): Int {  // [1]
  println("Inside multiplyByTwo") // [2]
  return x * 2
}

fun main() {
  val r = multiplyByTwo(5)        // [3]
  println(r)
}
/* Output:
Inside multiplyByTwo
10
*/
```

- **[1]** `fun` anahtar kelimesine, fonksiyon adına ve tek bir parametreden oluşan parametre listesine dikkat edin. Bu fonksiyon bir `Int` parametresi alır ve bir `Int` döner.
- **[2]** Bu iki satır fonksiyonun gövdesidir. Son satır, fonksiyonun sonucu olarak `x * 2` hesaplamasının değerini döner.
- **[3]** Bu satır, uygun bir argümanla fonksiyonu *çağırır* ve sonucu `val r` içine yakalar. Bir fonksiyon çağrısı, bildirisinin formunu taklit eder: Fonksiyon adı ve parantezler içinde argümanlar.

Fonksiyon kodu, fonksiyon adı `multiplyByTwo()` kullanılarak çağrılarak yürütülür, bu kodun bir kısaltması olarak. Bu nedenle fonksiyonlar, programlamada en temel basitleştirme ve kod yeniden kullanımı biçimidir. Ayrıca bir fonksiyonu, değiştirilebilir değerlere (parametrelere) sahip bir ifade olarak düşünebilirsiniz.

`println()` de bir fonksiyon çağrısıdır—sadece Kotlin tarafından sağlanmış bir fonksiyondur. Kotlin tarafından tanımlanan fonksiyonlara *kütüphane fonksiyonları* diyoruz.

Eğer fonksiyon anlamlı bir sonuç sağlamıyorsa, dönüş tipi `Unit` olur. İsterseniz `Unit`'i açıkça belirtebilirsiniz, ancak Kotlin bunu atlamanıza izin verir:

```
// Functions/SayHello.kt

fun sayHello() {
  println("Hallo!")
}

fun sayGoodbye(): Unit {
  println("Auf Wiedersehen!")
}

fun main() {
  sayHello()
  sayGoodbye()
}
/* Output:
Hallo!
Auf Wiedersehen!
*/
```

Hem `sayHello()` hem de `sayGoodbye()` fonksiyonları `Unit` döndürür, ancak `sayHello()` açıkça belirtmez. `main()` fonksiyonu da `Unit` döndürür.

Eğer bir fonksiyon sadece tek bir ifade içeriyorsa, eşittir işareti ve ardından gelen ifadeyi kullanarak kısaltılmış sözdizimini kullanabilirsiniz:

```
fun functionName(arg1: Type1, arg2: Type2, ...): ReturnType = expression
```

Küme parantezleriyle çevrili bir fonksiyon gövdesine *blok gövdesi* denir. Eşittir sözdizimini kullanan bir fonksiyon gövdesine *ifade gövdesi* denir.

Burada, `multiplyByThree()` bir ifade gövdesi kullanır:

```
// Functions/MultiplyByThree.kt

fun multiplyByThree(x: Int): Int = x * 3

fun main() {
  println(multiplyByThree(5))
}
/* Output:
15
*/
```

Bu, bir blok gövdesi içinde `return x * 3` demenin kısa bir versiyonudur.

Kotlin, ifade gövdesine sahip bir fonksiyonun dönüş tipini çıkarır:

```
// Functions/MultiplyByFour.kt

fun multiplyByFour(x: Int) = x * 4

fun main() {
  val result: Int = multiplyByFour(5)
  println(result)
}
/* Output:
20
*/
```

Kotlin, `multiplyByFour()` işlevinin bir `Int` döndüreceğini çıkarır.

Kotlin, yalnızca *ifade gövdelerinin* dönüş türlerini çıkarabilir. Bir işlevin bir blok gövdesi varsa ve türünü belirtmezseniz, o işlev `Unit` döndürür.

- -

Fonksiyon yazarken açıklayıcı isimler seçin. Bu, kodun okunmasını kolaylaştırır ve genellikle kod yorumlarına olan ihtiyacı azaltır. Bu kitapta fonksiyon isimleriyle ilgili olarak her zaman istediğimiz kadar açıklayıcı olamayabiliriz çünkü satır genişliğiyle sınırlıyız.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

`if` Deyimleri

> Bir *`if` deyimi* bir seçim yapar.

`if` anahtar kelimesi bir deyimi `true` veya `false` olup olmadığını test eder ve sonuca göre bir işlem gerçekleştirir. Doğru-yanlış ifadesine *Boolean* denir, bu ifadelerin ardındaki mantığı icat eden matematikçi George Boole'dan sonra bu isim verilmiştir. İşte > (büyüktür) ve < (küçüktür) sembollerini kullanan bir örnek:

```
// IfExpressions/If1.kt

fun main() {
  if (1 > 0)
    println("It's true!")
  if (10 < 11) {
    println("10 < 11")
    println("ten is less than eleven")
  }
}
/* Output:
It's true!
10 < 11
ten is less than eleven
*/
```

Parantez içindeki ifade `if`'den sonra `true` veya `false` olarak değerlendirilmelidir. Eğer `true` ise, sonraki ifade yürütülür. Birden fazla satır yürütmek için, onları küme parantezleri içine yerleştirin.

Bir yerde bir Boolean ifadesi oluşturabilir ve başka bir yerde kullanabiliriz:

```
// IfExpressions/If2.kt

fun main() {
  val x: Boolean = 1 >= 1
  if (x)
    println("It's true!")
}
/* Output:
It's true!
*/
```

`x` `Mantıksal` olduğundan, `if` doğrudan `if(x)` şeklinde test edebilir.

Mantıksal `>=` operatörü, operatörün sol tarafındaki ifade, sağ tarafındaki ifadeden *büyük veya eşit* ise `true` döner. Benzer şekilde, `<=` sol taraftaki ifade sağ taraftaki ifadeden *küçük veya eşit* ise `true` döner.

`else` anahtar kelimesi, hem `true` hem de `false` yollarını ele almanızı sağlar:

```
// IfExpressions/If3.kt

fun main() {
  val n: Int = -11
  if (n > 0)
    println("It's positive")
  else
    println("It's negative or zero")
}
/* Output:
It's negative or zero
*/
```

`else` anahtar kelimesi yalnızca `if` ile birlikte kullanılır. Tek bir kontrolle sınırlı değilsiniz—`else` ve `if` kombinasyonlarını birleştirerek birden fazla kombinasyonu test edebilirsiniz:

```
// IfExpressions/If4.kt

fun main() {
  val n: Int = -11
  if (n > 0)
    println("It's positive")
  else if (n == 0)
    println("It's zero")
  else
    println("It's negative")
}
/* Output:
It's negative
*/
```

Burada iki sayıyı eşitlik açısından kontrol etmek için == kullanıyoruz. != ise eşitsizlik için test eder.

Tipik desen, `if` ile başlayıp, ihtiyaç duyduğunuz kadar `else if` cümlesi ekleyip, önceki tüm testlerle eşleşmeyen her şey için son olarak bir `else` ile bitirmektir. Bir `if` ifadesi belirli bir boyuta ve karmaşıklığa ulaştığında muhtemelen bunun yerine bir `when` ifadesi kullanacaksınız. `when`, kitabın ilerleyen bölümlerinde, when İfadeleri bölümünde açıklanmıştır.

"not" operatörü `!`, bir Boolean ifadesinin tersini test eder:

```
// IfExpressions/If5.kt

fun main() {
  val y: Boolean = false
  if (!y)
    println("!y is true")
}
/* Output:
!y is true
*/
```

`if(!y)` ifadesini sözlü olarak ifade etmek için, "y değilse" deyin.

Tüm `if` bir ifadedir, bu yüzden bir sonuç üretebilir:

```
// IfExpressions/If6.kt

fun main() {
  val num = 10
  val result = if (num > 100) 4 else 42
  println(result)
}
/* Output:
42
*/
```

Burada, tüm `if` ifadesi tarafından üretilen değeri `result` adlı ara bir tanımlayıcıda saklıyoruz. Eğer koşul sağlanırsa, ilk dal `result` üretir. Sağlanmazsa, `else` değeri `result` olur.

Fonksiyonlar oluşturmayı pratik edelim. İşte bir Boolean parametre alan bir fonksiyon:

```
// IfExpressions/TrueOrFalse.kt

fun trueOrFalse(exp: Boolean): String {
  if (exp)
    return "It's true!"          // [1]
  return "It's false"            // [2]
}

fun main() {
  val b = 1
  println(trueOrFalse(b < 3))
  println(trueOrFalse(b >= 3))
}
/* Output:
It's true!
It's false
*/
```

Boolean parametre `exp` fonksiyonuna `trueOrFalse()` olarak geçer. Eğer argüman, `b < 3` gibi bir ifade olarak verilirse, bu ifade önce değerlendirilir ve sonuç fonksiyona iletilir. `trueOrFalse()` fonksiyonu `exp`'i test eder ve eğer sonuç `true` ise, **[1]** satırı çalıştırılır, aksi takdirde **[2]** satırı çalıştırılır.

- **[1]** `return` der ki, "Fonksiyondan çık ve bu değeri fonksiyonun sonucu olarak üret." Dikkat edin ki `return` bir fonksiyonun herhangi bir yerinde görünebilir ve sonunda olmak zorunda değildir.

Önceki örnekte olduğu gibi `return` kullanmak yerine, sonucu bir ifade olarak üretmek için `else` anahtar kelimesini kullanabilirsiniz:

```
// IfExpressions/OneOrTheOther.kt

fun oneOrTheOther(exp: Boolean): String =
  if (exp)
    "True!" // No 'return' necessary
  else
    "False"

fun main() {
  val x = 1
  println(oneOrTheOther(x == 1))
  println(oneOrTheOther(x == 2))
}
/* Output:
True!
False
*/
```

`trueOrFalse()` fonksiyonunda iki ifade yerine, `oneOrTheOther()` tek bir ifadedir. Bu ifadenin sonucu, fonksiyonun sonucudur, bu nedenle `if` ifadesi fonksiyon gövdesi haline gelir.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

String Şablonları

`String` şablonu, programatik olarak bir `String` oluşturmanın bir yoludur.

Bir tanımlayıcı adının önüne $ koyarsanız, `String` şablonu o tanımlayıcının içeriğini `String` içerisine yerleştirir:

```
// StringTemplates/StringTemplates.kt

fun main() {
  val answer = 42
  println("Found $answer!")     // [1]
  println("printing a $1")      // [2]
}
/* Output:
Found 42!
printing a $1
*/
```

- **[1]** `$answer` `answer` değerini yerine koyar.
- **[2]** $ işaretinden sonra gelen şey bir program tanımlayıcısı olarak tanınmazsa, özel bir şey olmaz.

Ayrıca, bir `String` içerisine değerleri birleştirme (+) kullanarak da ekleyebilirsiniz:

```
// StringTemplates/StringConcatenation.kt

fun main() {
  val s = "hi\n" // \n is a newline character
  val n = 11
  val d = 3.14
  println("first: " + s + "second: " +
    n + ", third: " + d)
}
/* Output:
first: hi
second: 11, third: 3.14
*/
```

Bir ifadeyi `${}` içine yerleştirmek, onu değerlendirir. Döndürdüğü değer bir `String`e dönüştürülür ve sonuçta oluşan `String`e eklenir:

```
// StringTemplates/ExpressionInTemplate.kt

fun main() {
  val condition = true
  println(
    "${if (condition) 'a' else 'b'}")  // [1]
  val x = 11
  println("$x + 4 = ${x + 4}")
}
/* Output:
a
11 + 4 = 15
*/
```

- **[1]** `if(condition) 'a' else 'b'` değerlendirilir ve sonucunda elde edilen değer tüm `${}` ifadesi yerine geçer.

Bir `String` özel bir karakter içermeli olduğunda, örneğin bir tırnak işareti gibi, bu karakteri bir `\` (*ters eğik çizgi*) ile kaçırtabilir veya üç tırnak içinde bir `String` sabiti kullanabilirsiniz:

```
// StringTemplates/TripleQuotes.kt

fun main() {
  val s = "value"
  println("s = \"$s\".")
  println("""s = "$s".""")
}
/* Output:
s = "value".
s = "value".
*/
```

Üçlü tırnak işaretleriyle, bir ifadenin değerini tek tırnaklı `String` ile yaptığınız gibi ekleyebilirsiniz.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Sayı Türleri

Farklı türde sayılar farklı şekillerde saklanır.

Bir tanımlayıcı oluşturur ve ona bir tamsayı değeri atarsanız, Kotlin `Int` türünü çıkarım yapar:

```
// NumberTypes/InferInt.kt

fun main() {
  val million = 1_000_000  // Infers Int
  println(million)
}
/* Output:
1000000
*/
```

Kotlin, okunabilirliği artırmak için sayısal değerlerin içinde alt çizgi kullanılmasına izin verir.

Sayılar için temel matematiksel operatörler, çoğu programlama dilinde bulunanlardır: toplama (+), çıkarma (-), bölme (/), çarpma (*) ve kalan bölme (%), bu da tam sayı bölmesinden kalanı üretir:

```
// NumberTypes/Modulus.kt

fun main() {
  val numerator: Int = 19
  val denominator: Int = 10
  println(numerator % denominator)
}
/* Output:
9
*/
```

Tam sayı bölmesi sonucunu keser:

```
// NumberTypes/IntDivisionTruncates.kt

fun main() {
  val numerator: Int = 19
  val denominator: Int = 10
  println(numerator / denominator)
}
/* Output:
1
*/
```

Eğer işlem sonucu yuvarlasaydı, çıktı `2` olurdu.

İşlem sırası temel aritmetiği takip eder:

```
// NumberTypes/OpOrder.kt

fun main() {
  println(45 + 5 * 6)
}
/* Output:
75
*/
```

Çarpma işlemi `5 * 6` önce gerçekleştirilir, ardından toplama `45 + 30` yapılır.

Eğer `45 + 5` işleminin önce gerçekleşmesini istiyorsanız, parantezler kullanın:

```
// NumberTypes/OpOrderParens.kt

fun main() {
  println((45 + 5) * 6)
}
/* Output:
300
*/
```

Şimdi *vücut kitle indeksini* (BMI) hesaplayalım, bu da kilonun kilogram cinsinden boyun metre cinsinden karesine bölünmesiyle elde edilir. Eğer BMI'niz `18.5`'ten azsa, zayıfsınız. `18.5` ile `24.9` arası normal kilodur. BMI'niz `25` veya daha yüksekse, fazla kilolusunuz. Bu örnek ayrıca, fonksiyonun parametrelerini tek bir satıra sığdıramadığınızda tercih edilen biçimlendirme stilini de göstermektedir:

```
// NumberTypes/BMIMetric.kt

fun bmiMetric(
  weight: Double,
  height: Double
): String {
  val bmi = weight / (height * height)  // [1]
  return if (bmi < 18.5) "Underweight"
    else if (bmi < 25) "Normal weight"
    else "Overweight"
}

fun main() {
  val weight = 72.57 // 160 lbs
  val height = 1.727 // 68 inches
  val status = bmiMetric(weight, height)
  println(status)
}
/* Output:
Normal weight
*/
```

- **[1]** Parantezleri çıkarırsanız, `weight`'i `height` ile böler, ardından bu sonucu `height` ile çarparsınız. Bu çok daha büyük bir sayı olur ve yanlış bir cevaptır.

`bmiMetric()`, ağırlık ve yükseklik için `Double` kullanır. Bir `Double`, çok büyük ve çok küçük kayan nokta sayıları tutar.

İngiliz birimlerini kullanarak, `Int` parametreleriyle temsil edilen bir versiyon:

```
// NumberTypes/BMIEnglish.kt

fun bmiEnglish(
  weight: Int,
  height: Int
): String {
  val bmi =
    weight / (height * height) * 703.07 // [1]
  return if (bmi < 18.5) "Underweight"
    else if (bmi < 25) "Normal weight"
    else "Overweight"
}
```

```
fun main() {
  val weight = 160
  val height = 68
  val status = bmiEnglish(weight, height)
  println(status)
}
/* Output:
Underweight
*/
```

Sonuç neden `Double` kullanan `bmiMetric()`'ten farklıdır? Bir tam sayıyı başka bir tam sayı ile böldüğünüzde, Kotlin bir tam sayı sonucu üretir. Tam sayı bölme sırasında kalanla başa çıkmanın standart yolu *kesmedir*, yani "kes ve at" (yuvarlama yoktur). Yani `5`'i `2`ye bölerseniz `2` elde edersiniz ve `7/10` sıfırdır. Kotlin, **[1]** ifadesindeki `bmi`yi hesapladığında, `160`ı `68 * 68` ile böler ve sıfır elde eder. Daha sonra sıfırı `703.07` ile çarparak sıfır elde eder.

Bu sorunu önlemek için, `703.07`yi hesaplamanın önüne taşıyın. Hesaplamalar bu şekilde `Double` olmaya zorlanır:

```
val bmi = 703.07 * weight / (height * height)
```

`bmiMetric()` içindeki `Double` parametreleri bu sorunu önler. Hesaplamaları mümkün olan en erken aşamada istenen türe dönüştürün ki doğruluk korunsun.

Tüm programlama dillerinin bir tamsayı içinde saklayabilecekleri sınırları vardır. Kotlin'in `Int` türü, -2^{31} ile $+2^{31}-1$ arasında değerler alabilir, bu `Int` 32-bit temsiliyetinin bir kısıtlamasıdır. Eğer yeterince büyük iki `Int`'i toplar veya çarparsanız, sonucu taşırırsınız:

```
// NumberTypes/IntegerOverflow.kt

fun main() {
  val i: Int = Int.MAX_VALUE
  println(i + i)
}
/* Output:
-2
*/
```

`Int.MAX_VALUE`, bir `Int`'in tutabileceği en büyük değerdir.

Taşma, beklediğimizden çok daha küçük ve negatif olduğu için açıkça yanlış bir sonuç üretir. Kotlin, potansiyel bir taşma algıladığında her zaman bir uyarı verir.

Taşmayı önlemek sizin, yani geliştiricinin sorumluluğundadır. Kotlin, derleme sırasında her zaman taşmayı algılayamaz ve taşmayı önlemez çünkü bu kabul edilemez bir performans etkisi yaratır.

Programınız büyük sayılar içeriyorsa, -2^{63} ile $+2^{63}-1$ arasındaki değerleri barındıran `Long`ları kullanabilirsiniz. Bir `Long` türünde bir `val` tanımlamak için türü açıkça belirtebilir veya sayısal bir literalin sonuna `L` koyarak Kotlin'in o değeri `Long` olarak kabul etmesini sağlayabilirsiniz:

```
// NumberTypes/LongConstants.kt

fun main() {
  val i = 0          // Infers Int
  val l1 = 0L        // L creates Long
  val l2: Long = 0   // Explicit type
  println("$l1 $l2")
}
/* Output:
0 0
*/
```

`Long`'lara geçerek `IntegerOverflow.kt`'deki taşmayı önlüyoruz:

```
// NumberTypes/UsingLongs.kt

fun main() {
  val i = Int.MAX_VALUE
  println(0L + i + i)              // [1]
  println(1_000_000 * 1_000_000L)  // [2]
}
/* Output:
4294967294
1000000000000
*/
```

Hem **[1]** hem de **[2]**'de sayısal bir literal kullanmak `Long` hesaplamaları zorlar ve ayrıca `Long` türünde bir sonuç üretir. `L`'nin göründüğü yer önemsizdir. Değerlerden biri `Long` ise, ortaya çıkan ifade `Long` olur.

`Int`'lerden çok daha büyük değerler tutabilmelerine rağmen, `Long`'ların da boyut sınırlamaları vardır:

```
// NumberTypes/BiggestLong.kt

fun main() {
  println(Long.MAX_VALUE)
}
/* Output:
9223372036854775807
*/
```

`Long.MAX_VALUE` bir `Long`'un alabileceği en büyük değerdir.

Egzersizler ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Booleans

> `if` İfadeleri, bir `Boolean` değerini tersine çeviren "not" operatörü `!`'yi gösterdi. Bu atom, daha fazla *Boolean Cebiri* tanıtır.

"ve" ve "veya" operatörleri ile başlıyoruz:

- `&&` (ve): Operatörün solunda ve sağında bulunan `Boolean` ifadeleri her ikisi de `true` olduğunda `true` üretir.
- `||` (veya): Operatörün solunda veya sağında bulunan ifadelerden biri veya her ikisi `true` olduğunda `true` üretir.

Bu örnekte, bir işletmenin açık mı kapalı mı olduğunu `saate` göre belirliyoruz:

```
// Booleans/Open1.kt

fun isOpen1(hour: Int) {
  val open = 9
  val closed = 20
  println("Operating hours: $open - $closed")
  val status =
    if (hour >= open && hour < closed) // [1]
      true
    else
      false
  println("Open: $status")
}

fun main() = isOpen1(6)
/* Output:
Operating hours: 9 - 20
Open: false
*/
```

`main()` tek bir fonksiyon çağrısıdır, bu yüzden Fonksiyonlar bölümünde anlatıldığı gibi bir ifade gövdesi kullanabiliriz.

[1] numaralı `if` ifadesi, `hour` değişkeninin açılış saati ile kapanış saati arasında olup olmadığını kontrol eder, bu yüzden ifadeleri `Boolean` `&&` (ve) operatörü ile birleştiririz.

`if` ifadesi basitleştirilebilir. `if(cond) true else false` ifadesinin sonucu sadece `cond` ifadesidir:

```
// Booleans/Open2.kt

fun isOpen2(hour: Int) {
  val open = 9
  val closed = 20
  println("Operating hours: $open - $closed")
  val status = hour >= open && hour < closed
  println("Open: $status")
}

fun main() = isOpen2(6)
/* Output:
Operating hours: 9 - 20
Open: false
*/
```

Mantığı tersine çevirelim ve iş yerinin şu anda kapalı olup olmadığını kontrol edelim. "veya" operatörü `||` en az bir koşul sağlanırsa `true` değerini üretir:

```
// Booleans/Closed.kt

fun isClosed(hour: Int) {
  val open = 9
  val closed = 20
  println("Operating hours: $open - $closed")
  val status = hour < open || hour >= closed
  println("Closed: $status")
}

fun main() = isClosed(6)
/* Output:
Operating hours: 9 - 20
Closed: true
*/
```

`Boolean` operatörler, karmaşık mantığı kompakt ifadelerde mümkün kılar. Ancak, işler kolayca kafa karıştırıcı hale gelebilir. Okunabilirlik için çaba gösterin ve niyetlerinizi açıkça belirtin.

İşte farklı değerlendirme sırasının farklı sonuçlar ürettiği karmaşık bir `Boolean` ifadesine bir örnek:

```
// Booleans/EvaluationOrder.kt

fun main() {
  val sunny = true
  val hoursSleep = 6
  val exercise = false
  val temp = 55

  // [1]:
  val happy1 = sunny && temp > 50 ||
    exercise && hoursSleep > 7
  println(happy1)

  // [2]:
  val sameHappy1 = (sunny && temp > 50) ||
    (exercise && hoursSleep > 7)
  println(sameHappy1)

  // [3]:
  val notSame =
    (sunny && temp > 50 || exercise) &&
      hoursSleep > 7
  println(notSame)
}
/* Output:
true
true
false
*/
```

`Boolean` ifadeleri `sunny`, `temp > 50`, `exercise` ve `hoursSleep > 7`. `happy1` ifadesini "Güneşli *ve* sıcaklık 50'nin üzerinde *veya* egzersiz yaptım *ve* 7 saatten fazla uyudum" olarak okuruz. Ama `&&` `||`'den önce mi gelir, yoksa tersi mi?

[1] numaralı ifadede Kotlin'in varsayılan değerlendirme sırası kullanılır. Bu, **[2]** numaralı ifadeyle aynı sonucu üretir çünkü parantezler olmadan, önce "ve"ler, sonra

"veya"lar değerlendirilir. **[3]** numaralı ifade, farklı bir sonuç üretmek için parantezler kullanır. **[3]** numaralı ifadede, sadece en az 7 saat uyuduysak mutlu oluruz.

Egzersizler ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

`while` ile Tekrar

> Bilgisayarlar tekrarlayan görevler için idealdir.

En temel tekrar biçimi `while` anahtar kelimesini kullanır. Bu, kontrol eden *Boolean ifadesi* `true` olduğu sürece bir bloğu tekrar eder:

```
while (Boolean-expression) {
  // Code to be repeated
}
```

Boolean ifadesi döngünün başında bir kez ve blok boyunca her bir yinelemeden önce tekrar değerlendirilir.

```
// RepetitionWithWhile/WhileLoop.kt

fun condition(i: Int) = i < 100  // [1]

fun main() {
  var i = 0
  while (condition(i)) {         // [2]
    print(".")
    i += 10                      // [3]
  }
}
/* Output:
..........
*/
```

- **[1]** Karşılaştırma operatörü `<` bir `Boolean` sonucu üretir, bu yüzden Kotlin `condition()` için `Boolean` sonucunu çıkarır.
- **[2]** `while` için koşullu ifade şöyle der: "gövdedeki ifadeleri `condition()` `true` döndüğü sürece tekrarla."
- **[3]** `+ =` operatörü `i`'ye `10` ekler ve sonucu tek bir işlemde `i`'ye atar (bu işlemin çalışması için `i` bir `var` olmalıdır). Bu aşağıdakine eşdeğerdir:

```
i = i + 10
```

while döngüsünü do anahtar kelimesi ile birlikte kullanmanın ikinci bir yolu daha vardır:

```
do {
  // Code to be repeated
} while (Boolean-expression)
```

WhileLoop.kt dosyasını do-while kullanacak şekilde yeniden yazmak:

```
// RepetitionWithWhile/DoWhileLoop.kt

fun main() {
  var i = 0
  do {
    print(".")
    i += 10
  } while (condition(i))
}
/* Output:
..........
*/
```

while ve do-while arasındaki tek fark, do-while döngüsünün gövdesinin, Boolean ifadesi başlangıçta false (yanlış) ürettiğinde bile en az bir kez çalışmasıdır. while döngüsünde, koşul ilk seferde false (yanlış) ise, gövde hiç çalışmaz. Pratikte, do-while, while döngüsünden daha az yaygındır.

Kısa versiyonlu atama operatörleri tüm aritmetik işlemler için mevcuttur: + =, -=, * =, / =, ve % =. Bu -= ve % = kullanır:

```
// RepetitionWithWhile/AssignmentOperators.kt

fun main() {
  var n = 10
  val d = 3
  print(n)
  while (n > d) {
    n -= d
    print(" - $d")
  }
  println(" = $n")

  var m = 10
  print(m)
  m %= d
  println(" % $d = $m")
}
/* Output:
10 - 3 - 3 - 3 = 1
10 % 3 = 1
*/
```

İki doğal sayının tam sayı bölme kalanını hesaplamak için, bir `while` döngüsü ile başlıyoruz, ardından kalan operatörünü kullanıyoruz.

Bir sayıya `1` eklemek ve `1` çıkarmak o kadar yaygındır ki, kendi artırma ve azaltma operatörlerine sahiptirler: `++` ve `--`. `i += 1` ifadesini `i++` ile değiştirebilirsiniz:

```
// RepetitionWithWhile/IncrementOperator.kt

fun main() {
  var i = 0
  while (i < 4) {
    print(".")
    i++
  }
}
/* Output:
....
*/
```

Uygulamada, `while` döngüleri bir sayı aralığında yineleme yapmak için kullanılmaz. Bunun yerine `for` döngüsü kullanılır. Bu, sonraki atomda ele alınmıştır.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Döngüler ve Aralıklar

`for` anahtar kelimesi, bir dizideki her değer için bir kod bloğu çalıştırır.

Değerler kümesi bir tamsayı aralığı, bir `String` veya kitabın ilerleyen bölümlerinde göreceğiniz gibi bir öğe koleksiyonu olabilir. `in` anahtar kelimesi, değerler arasında adım attığınızı belirtir:

```
for (v in values) {
  // Do something with v
}
```

Her döngüde, `v` değişkenine `values` içindeki bir sonraki eleman atanır.

İşte belirli bir sayıda eylemi tekrarlayan bir `for` döngüsü:

```
// LoopingAndRanges/RepeatThreeTimes.kt

fun main() {
  for (i in 1..3) {
    println("Hey $i!")
  }
}
/* Output:
Hey 1!
Hey 2!
Hey 3!
*/
```

Çıktı, dizin `i`'nin 1'den 3'e kadar olan aralıktaki her değeri aldığını gösterir.

Bir *aralık*, bir çift uç nokta tarafından tanımlanan bir değer aralığıdır. Aralıkları tanımlamanın iki temel yolu vardır:

```
// LoopingAndRanges/DefiningRanges.kt

fun main() {
  val range1 = 1..10          // [1]
  val range2 = 0 until 10     // [2]
  println(range1)
  println(range2)
}
/* Output:
1..10
0..9
*/
```

- **[1]** `..` sözdiziminin kullanılması, sonuçtaki aralığa her iki sınırı da dahil eder.
- **[2]** `until` sonu hariç tutar. Çıktı, `10`'un aralığın bir parçası olmadığını gösterir.

Bir aralığın görüntülenmesi okunabilir bir format üretir.

Bu, 10'dan 100'e kadar olan sayıları toplar:

```
// LoopingAndRanges/SumUsingRange.kt

fun main() {
  var sum = 0
  for (n in 10..100) {
    sum += n
  }
  println("sum = $sum")
}
/* Output:
sum = 5005
*/
```

Bir aralık üzerinde ters sırayla yineleyebilirsiniz. Ayrıca, varsayılan `1` değerinden farklı bir aralık kullanmak için bir `adım` değeri de kullanabilirsiniz:

```
// LoopingAndRanges/ForWithRanges.kt

fun showRange(r: IntProgression) {
  for (i in r) {
    print("$i ")
  }
  print("    // $r")
  println()
}

fun main() {
  showRange(1..5)
  showRange(0 until 5)
  showRange(5 downTo 1)          // [1]
  showRange(0..9 step 2)         // [2]
  showRange(0 until 10 step 3)   // [3]
  showRange(9 downTo 2 step 3)
}
/* Output:
1 2 3 4 5     // 1..5
0 1 2 3 4     // 0..4
5 4 3 2 1     // 5 downTo 1 step 1
0 2 4 6 8     // 0..8 step 2
0 3 6 9     // 0..9 step 3
9 6 3     // 9 downTo 3 step 3
*/
```

- **[1]** `downTo` azalan bir aralık üretir.
- **[2]** `step` aralığı değiştirir. Burada, aralık bir yerine iki değerle adımlanır.
- **[3]** `until step` ile birlikte de kullanılabilir. Bunun çıktıyı nasıl etkilediğine dikkat edin.

Her durumda sayılar dizisi bir aritmetik dizi oluşturur. `showRange()` bir `IntProgression` parametresi kabul eder, bu `Int` aralıklarını içeren yerleşik bir türdür. Her satırın çıktı yorumunda görünen her `IntProgression`'ın `String` gösterimine dikkat edin, bu genellikle `showRange()`'a geçirilen aralıktan farklıdır—`IntProgression` girişi eşdeğer ortak bir forma dönüştürmektedir.

Ayrıca bir karakter aralığı da üretebilirsiniz. Bu `for` `a`'dan `z`'ye kadar yinelemektedir:

```
// LoopingAndRanges/ForWithCharRange.kt

fun main() {
  for (c in 'a'..'z') {
    print(c)
  }
}
/* Output:
abcdefghijklmnopqrstuvwxyz
*/
```

Tam sayılar ve karakterler gibi bütünsel miktarları içeren bir aralık üzerinde yineleme yapabilirsiniz, ancak kayan noktalı değerler için bu geçerli değildir.

Köşeli parantezler karakterlere indeks ile erişir. `String` içindeki karakterleri sıfırdan saymaya başladığımız için, `s[0]`, `String s`'in ilk karakterini seçer. `s.lastIndex`'i seçmek, son indeks numarasını üretir:

```
// LoopingAndRanges/IndexIntoString.kt

fun main() {
  val s = "abc"
  for (i in 0..s.lastIndex) {
    print(s[i] + 1)
  }
}
/* Output:
bcd
*/
```

Bazen insanlar `s[0]` için "sıfırıncı karakter" derler.

Karakterler, Unicode[17] değerlerine karşılık gelen sayılar olarak saklanır, bu nedenle bir karaktere bir tam sayı eklemek, yeni kod değerine karşılık gelen yeni bir karakter üretir:

[17] https://en.wikipedia.org/wiki/Unicode

```
// LoopingAndRanges/AddingIntToChar.kt

fun main() {
  val ch: Char = 'a'
  println(ch + 25)
  println(ch < 'z')
}
/* Output:
z
true
*/
```

İkinci `println()`, karakter kodlarını karşılaştırabileceğinizi gösterir.

Bir `for` döngüsü doğrudan `String`ler üzerinde yineleme yapabilir:

```
// LoopingAndRanges/IterateOverString.kt

fun main() {
  for (ch in "Jnskhm ") {
    print(ch + 1)
  }
}
/* Output:
Kotlin!
*/
```

`ch` her karakteri sırayla alır.

Aşağıdaki örnekte, `hasChar()` fonksiyonu `String` `s` üzerinde iterasyon yapar ve belirli bir karakter `ch` içerip içermediğini test eder. Fonksiyonun ortasındaki `return`, cevap bulunduğunda fonksiyonu durdurur:

```
// LoopingAndRanges/HasChar.kt

fun hasChar(s: String, ch: Char): Boolean {
  for (c in s) {
    if (c == ch) return true
  }
  return false
}

fun main() {
  println(hasChar("kotlin", 't'))
  println(hasChar("kotlin", 'a'))
}
/* Output:
true
false
*/
```

Sonraki atom, `hasChar()` fonksiyonunun gereksiz olduğunu gösteriyor—yerine yerleşik sözdizimini kullanabilirsiniz.

Eğer bir eylemi belirli bir sayıda tekrarlamak istiyorsanız, bir `for` döngüsü yerine `repeat()` fonksiyonunu kullanabilirsiniz:

```
// LoopingAndRanges/RepeatHi.kt

fun main() {
  repeat(2) {
    println("hi!")
  }
}
/* Output:
hi!
hi!
*/
```

`repeat()` standart bir kütüphane fonksiyonudur, anahtar kelime değildir. Kitabın ilerleyen bölümlerinde nasıl oluşturulduğunu göreceksiniz.

Alıştırmalar ve çözümler şu adreste bulunabilir: www.AtomicKotlin.com.

`in` Anahtar Kelimesi

`in` anahtar kelimesi, bir değerin bir aralıkta olup olmadığını test eder.

```
// InKeyword/MembershipInRange.kt

fun main() {
  val percent = 35
  println(percent in 1..100)
}
/* Output:
true
*/
```

Booleans bölümünde sınırları açıkça kontrol etmeyi öğrendiniz:

```
// InKeyword/MembershipUsingBounds.kt

fun main() {
  val percent = 35
  println(0 <= percent && percent <= 100)
}
/* Output:
true
*/
```

`0 <= x && x <= 100` mantıksal olarak `x in 0..100` ifadesine eşdeğerdir. IntelliJ IDEA, ilk biçimi otomatik olarak ikinci biçimle değiştirmeyi önerir, bu da okunması ve anlaşılması daha kolaydır.

`in` anahtar kelimesi hem yineleme hem de üyelik için kullanılır. Bir `for` döngüsünün kontrol ifadesindeki bir `in`, yineleme anlamına gelir, aksi takdirde `in` üyeliği kontrol eder:

```
// InKeyword/IterationVsMembership.kt

fun main() {
  val values = 1..3
  for (v in values) {
    println("iteration $v")
  }
  val v = 2
  if (v in values)
    println("$v is a member of $values")
}
/* Output:
iteration 1
iteration 2
iteration 3
2 is a member of 1..3
*/
```

`in` anahtar kelimesi sadece aralıklarla sınırlı değildir. Bir karakterin bir `String`in parçası olup olmadığını da kontrol edebilirsiniz. Aşağıdaki örnek, önceki atomdan `hasChar()` yerine `in` kullanmaktadır:

```
// InKeyword/InString.kt

fun main() {
  println('t' in "kotlin")
  println('a' in "kotlin")
}
/* Output:
true
false
*/
```

Kitabın ilerleyen bölümlerinde `in` ifadesinin başka türlerle de çalıştığını göreceksiniz.

Burada, `in` bir karakterin bir karakter aralığına ait olup olmadığını test eder:

```
// InKeyword/CharRange.kt

fun isDigit(ch: Char) = ch in '0'..'9'

fun notDigit(ch: Char) =
  ch !in '0'..'9'               // [1]

fun main() {
  println(isDigit('a'))
  println(isDigit('5'))
  println(notDigit('z'))
}
/* Output:
false
true
true
*/
```

- **[1]** `!in` bir değerin bir aralığa ait olmadığını kontrol eder.

Bir `Double` aralığı oluşturabilirsiniz, ancak bunu yalnızca üyelik kontrolü için kullanabilirsiniz:

```
// InKeyword/FloatingPointRange.kt

fun inFloatRange(n: Double) {
  val r = 1.0..10.0
  println("$n in $r? ${n in r}")
}

fun main() {
  inFloatRange(0.999999)
  inFloatRange(5.0)
  inFloatRange(10.0)
  inFloatRange(10.0000001)
}
/* Output:
0.999999 in 1.0..10.0? false
5.0 in 1.0..10.0? true
10.0 in 1.0..10.0? true
10.0000001 in 1.0..10.0? false
*/
```

Kotlin'de bir ondalık aralığı tanımlamak için yalnızca `..` kullanabilirsiniz.

Bir `String`'in bir `String` aralığının üyesi olup olmadığını kontrol edebilirsiniz:

```
// InKeyword/StringRange.kt

fun main() {
  println("ab" in "aa".."az")
  println("ba" in "aa".."az")
}
/* Output:
true
false
*/
```

Burada Kotlin alfabetik karşılaştırma kullanır.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

İfadeler & Açıklamalar

> *Açıklamalar* ve *ifadeler* çoğu programlama dilinde en küçük faydalı kod parçalarıdır.

Temel bir fark vardır: bir açıklama bir etki yaratır, ancak bir sonuç üretmez. Bir ifade ise her zaman bir sonuç üretir.

Sonuç üretmediği için, bir açıklama çevresinin durumunu değiştirmelidir. Başka bir deyişle, "bir açıklama *yan etkileri* için çağrılır" (yani, bir sonuç üretmek *dışında* ne yaptığı). Hafıza yardımı olarak:

> *Bir açıklama durumu değiştirir.*

"İfade etmek" tanımlarından biri "zorlamak veya sıkmak"tır, örneğin "bir portakalın suyunu ifade etmek" gibi. Yani

> *Bir ifade ifade eder.*

Yani, bir sonuç üretir.

`for` döngüsü Kotlin'de bir açıklamadır. Bir sonuç olmadığı için onu atayamazsınız:

```
// ExpressionsStatements/ForIsAStatement.kt

fun main() {
  // Can't do this:
  // val f = for(i in 1..10) {}
  // Compiler error message:
  // for is not an expression, and
  // only expressions are allowed here
}
```

Bir `for` döngüsü yan etkileri için kullanılır.

Bir ifade bir değer üretir, bu değer atanabilir veya başka bir ifadenin parçası olarak kullanılabilir, oysa bir bildirim her zaman üst düzey bir öğedir.

Her fonksiyon çağrısı bir ifadedir. Fonksiyon `Unit` döndürse ve sadece yan etkileri için çağrılsa bile, sonuç yine de atanabilir:

```
// ExpressionsStatements/UnitReturnType.kt

fun unitFun() = Unit

fun main() {
  println(unitFun())
  val u1: Unit = println(42)
  println(u1)
  val u2 = println(0) // Type inference
  println(u2)
}
/* Output:
kotlin.Unit
42
kotlin.Unit
0
kotlin.Unit
*/
```

`Unit` türü, `Unit` adı verilen tek bir değer içerir ve bu değeri doğrudan, `unitFun()` içinde görüldüğü gibi döndürebilirsiniz. `println()` çağrısı da `Unit` döndürür. `val u1`, `println()`'ın dönüş değerini yakalar ve açıkça `Unit` olarak tanımlanır, oysa `u2` tip çıkarımı kullanır.

`if` bir ifade oluşturur, bu yüzden sonucunu atayabilirsiniz:

```
// ExpressionsStatements/AssigningAnIf.kt

fun main() {
  val result1 = if (11 > 42) 9 else 5

  val result2 = if (1 < 2) {
    val a = 11
    a + 42
  } else 42

  val result3 =
    if ('x' < 'y')
      println("x < y")
    else
      println("x > y")

  println(result1)
  println(result2)
  println(result3)
}
/* Output:
x < y
5
53
kotlin.Unit
*/
```

İlk çıkan satır `x < y`'dir, `result3` `main()`'in sonunda görüntülense bile. Bu, `result3` değerlendirilirken `println()` çağrıldığı için olur ve değerlendirme `result3` tanımlandığında gerçekleşir.

Dikkat edin, `a` `result2` kod bloğu içinde tanımlanmıştır. Son ifadenin sonucu, `if` ifadesinin sonucu olur; burada, 11 ve 42'nin toplamıdır. Peki ya `a`? Kod bloğundan çıktığınızda (küme parantezlerinden dışarı çıktığınızda), `a`'ya erişemezsiniz. O, *geçici*dir ve bu bloğun *kapsamı*ndan çıktığınızda atılır.

Artış operatörü `i++` da bir ifadedir, bir ifade gibi görünse bile. Kotlin, C gibi dillerin kullandığı yaklaşımı takip eder ve biraz farklı anlamlara sahip iki versiyon artış ve azalış operatörleri sağlar. Önek operatörü operandın önüne gelir, örneğin `++i` ve artış gerçekleştikten sonra değeri döner. Bunu "önce artışı yap, sonra ortaya çıkan değeri döndür" olarak okuyabilirsiniz. Sonek operatörü ise operandın arkasına yerleştirilir,

örneğin `i++` ve artış gerçekleşmeden önce `i`'nin değerini döner. Bunu da "önce sonucu üret, sonra artışı yap" olarak okuyabilirsiniz.

```
// ExpressionsStatements/PostfixVsPrefix.kt

fun main() {
  var i = 10
  println(i++)
  println(i)
  var j = 20
  println(++j)
  println(j)
}
/* Output:
10
11
21
21
*/
```

Çıkma operatörünün de iki versiyonu vardır: `--i` ve `i--`. Artırma ve çıkma operatörlerini diğer ifadeler içinde kullanmak tavsiye edilmez, çünkü kafa karıştırıcı kod üretebilir:

```
// ExpressionsStatements/Confusing.kt

fun main() {
  var i = 1
  println(i++ + ++i)
}
```

Çıktının ne olacağını tahmin etmeye çalışın, sonra kontrol edin.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Özet 1

> Bu atom, Bölüm I'deki Merhaba, Dünya! ile başlayıp İfadeler ve Deyimler ile biten atomları özetler ve inceler.

Eğer deneyimli bir programcıysanız, bu sizin ilk atomunuz olmalıdır. Yeni programcılar, Bölüm I'in gözden geçirilmesi için bu atomu okumalı ve alıştırmaları yapmalıdır.

Eğer herhangi bir şey açık değilse, o konuyla ilgili atomu inceleyin (alt başlıklar atom başlıklarına karşılık gelir).

Merhaba, Dünya

Kotlin, hem `//` tek satırlık yorumları hem de `/*`-den `*/`-e kadar çok satırlı yorumları destekler. Bir programın giriş noktası `main()` fonksiyonudur:

```
// Summary1/Hello.kt

fun main() {
  println("Hello, world!")
}
/* Output:
Hello, world!
*/
```

Bu kitaptaki her örneğin ilk satırı, atomun alt dizin adını içeren bir yorum ve ardından bir `/` ve dosya adını içerir. Tüm çıkarılan kod örneklerini AtomicKotlin.com[18] adresinde bulabilirsiniz.

`println()`, tek bir `String` parametresi (veya bir `String`'e dönüştürülebilen bir parametre) alan standart bir kütüphane fonksiyonudur. `println()`, parametresini görüntüledikten sonra imleci yeni bir satıra taşır, `print()` ise imleci aynı satırda bırakır.

Kotlin, bir ifade veya deyimin sonunda noktalı virgül gerektirmez. Noktalı virgüller yalnızca tek bir satırda birden fazla ifade veya deyimi ayırmak için gereklidir.

[18] http://AtomicKotlin.com

`var` & `val`, Veri Türleri

Değişmeyen bir tanımlayıcı oluşturmak için, `val` anahtar sözcüğünü, ardından tanımlayıcı adını, iki nokta üst üste ve bu değerin türünü kullanın. Sonra bir eşittir işareti ve bu `val`'e atamak istediğiniz değeri ekleyin:

```
val identifier: Type = initialization
```

Bir kez `val` atandığında, yeniden atanamaz.

Kotlin'in *type inference* genellikle tipi başlangıç değerine dayanarak otomatik olarak belirleyebilir. Bu, daha basit bir tanım üretir:

```
val identifier = initialization
```

Aşağıdakilerin her ikisi de geçerlidir:

```
val daysInFebruary = 28
val daysInMarch: Int = 31
```

Bir `var` (değişken) tanımı, `val` yerine `var` kullanılarak aynı görünür:

```
var identifier1 = initialization
var identifier2: Type = initialization
```

`val`'in aksine, bir `var`'ı değiştirebilirsiniz, bu nedenle aşağıdaki yasaldır:

```
var hoursSpent = 20
hoursSpent = 25
```

Ancak, *type* değiştirilemez, bu yüzden şöyle derseniz bir hata alırsınız:

```
hoursSpent = 30.5
```

Kotlin `hoursSpent` tanımlandığında `Int` türünü çıkarımlar, bu nedenle ondalık bir değere değişimi kabul etmez.

Fonksiyonlar

Fonksiyonlar adlandırılmış alt programlardır:

```
fun functionName(arg1: Type1, arg2: Type2, ...): ReturnType {
  // Lines of code ...
  return result
}
```

`fun` anahtar kelimesi, fonksiyon adı ve parantez içinde parametre listesi ile takip edilir. Her parametrenin açık bir türü olmalıdır çünkü Kotlin parametre türlerini çıkaramaz. Fonksiyonun kendisi, `var` veya `val` için olduğu gibi (bir iki nokta üst üste ve ardından tür) tanımlanmış bir türe sahiptir. Bir fonksiyonun türü, döndürülen sonucun türüdür.

Fonksiyon imzasını, küme parantezleri içinde yer alan fonksiyon gövdesi takip eder. `return` ifadesi fonksiyonun dönüş değerini sağlar.

Fonksiyon tek bir ifadeden oluşuyorsa, kısaltılmış bir sözdizimi kullanabilirsiniz:

```
fun functionName(arg1: Type1, arg2: Type2, ...): ReturnType = result
```

Bu form *ifade gövdesi* olarak adlandırılır. Açılış süslü parantez kullanmak yerine, ifadenin ardından bir eşittir işareti kullanın. Kotlin dönüş türünü çıkardığı için dönüş türünü belirtmenize gerek yoktur.

İşte parametresinin küpünü üreten bir fonksiyon ve bir `String`'e ünlem işareti ekleyen başka bir fonksiyon:

```
// Summary1/BasicFunctions.kt

fun cube(x: Int): Int {
  return x * x * x
}

fun bang(s: String) = s + "!"

fun main() {
  println(cube(3))
  println(bang("pop"))
}
/* Output:
27
pop!
*/
```

`cube()` açık bir `return` ifadesi ile bir blok gövdesine sahiptir. `bang()` fonksiyonun dönüş değerini üreten bir ifade gövdesidir. Kotlin, `bang()`'in dönüş türünü `String` olarak çıkarır.

Booleans

Boole Cebiri için, Kotlin şu operatörleri sağlar:

- `!` (değil), değeri mantıksal olarak tersine çevirir (`true`'yi `false` yapar ve tam tersi).
- `&&` (ve), yalnızca *her iki* koşul da `true` ise `true` döner.
- `||` (veya), koşullardan en az biri `true` ise `true` döner.

```
// Summary1/Booleans.kt

fun main() {
  val opens = 9
  val closes = 20
  println("Operating hours: $opens - $closes")
  val hour = 6
  println("Current time: " + hour)

  val isOpen = hour >= opens && hour < closes
  println("Open: " + isOpen)
  println("Not open: " + !isOpen)

  val isClosed = hour < opens || hour >= closes
  println("Closed: " + isClosed)
}
/* Output:
Operating hours: 9 - 20
Current time: 6
Open: false
Not open: true
Closed: true
*/
```

`isOpen`'in başlatıcısı, her iki koşulun da `true` olup olmadığını test etmek için `&&` kullanır. İlk koşul `hour >= opens`, `false` olduğu için tüm ifadenin sonucu `false` olur. `isClosed` için başlatıcı ise `||` kullanarak, koşullardan en az biri `true` olduğunda `true` üretir. İfade `hour < opens` `true` olduğu için tüm ifade `true` olur.

`if` İfadeleri

`if` bir ifade olduğundan, bir sonuç üretir. Bu sonuç bir `var` veya `val`'a atanabilir. Burada ayrıca `else` anahtar kelimesinin kullanımını da görebilirsiniz:

```
// Summary1/IfResult.kt

fun main() {
  val result = if (99 < 100) 4 else 42
  println(result)
}
/* Output:
4
*/
```

`if` ifadesinin her iki dalı da süslü parantezlerle çevrili çok satırlı bir kod bloğu olabilir:

```
// Summary1/IfExpression.kt

fun main() {
  val activity = "swimming"
  val hour = 10

  val isOpen = if (
    activity == "swimming" ||
    activity == "ice skating") {
    val opens = 9
    val closes = 20
    println("Operating hours: " +
      opens + " - " + closes)
    hour >= opens && hour < closes
  } else {
    false
  }
  println(isOpen)
}
/* Output:
Operating hours: 9 - 20
true
*/
```

Bir kod bloğu içinde tanımlanan bir değer, örneğin `opens`, o bloğun kapsamı dışında erişilebilir değildir. Çünkü `if` ifadesine *küresel olarak* tanımlanmışlardır, `activity` ve `hour` `if` ifadesi içinde erişilebilirdir.

Bir `if` ifadesinin sonucu, seçilen dalın son ifadesinin sonucudur. Burada, `hour >= opens && hour <= closes` ifadesi `true` olur.

String Şablonları

Bir `String` içinde bir değeri `String` şablonları kullanarak ekleyebilirsiniz. Tanımlayıcı adının önüne bir $ ekleyin:

```
// Summary1/StrTemplates.kt

fun main() {
  val answer = 42
  println("Found $answer!")            // [1]
  val condition = true
  println(
    "${if (condition) 'a' else 'b'}")  // [2]
  println("printing a $1")             // [3]
}
/* Output:
Found 42!
a
printing a $1
*/
```

- **[1]** `$answer`, `answer` içinde bulunan değeri yerine koyar.
- **[2]** `${if(condition) 'a' else 'b'}`, `${}` içindeki ifadenin sonucunu değerlendirir ve yerine koyar.
- **[3]** Eğer $ bir program tanımlayıcısı olarak tanınmayan herhangi bir şey tarafından takip edilirse, özel bir şey olmaz.

Çok satırlı metni veya özel karakterleri saklamak için üçlü tırnak işaretli `String`'leri kullanın:

```
// Summary1/ThreeQuotes.kt

fun json(q: String, a: Int) = """{
  "question" : "$q",
  "answer" : $a
}"""

fun main() {
  println(json("The Ultimate", 42))
}
/* Output:
{
  "question" : "The Ultimate",
  "answer" : 42
}
*/
```

Üç tırnaklı bir `String` içinde `"` gibi özel karakterleri kaçışa gerek yoktur. (Normal bir `String` içinde çift tırnak eklemek için `\"` yazarsınız). Normal `String`lerde olduğu gibi, üç tırnaklı bir `String` içinde `$` kullanarak bir tanımlayıcı veya bir ifade ekleyebilirsiniz.

Sayı Türleri

Kotlin, tamsayı türleri (`Int`, `Long`) ve ondalık sayı türleri (`Double`) sağlar. Bir tam sayı sabiti varsayılan olarak `Int`'tir ve sonuna `L` eklenirse `Long` olur. Bir sabit, ondalık nokta içeriyorsa `Double` olur:

```
// Summary1/NumberTypes.kt

fun main() {
  val n = 1000    // Int
  val l = 1000L   // Long
  val d = 1000.0  // Double
  println("$n $l $d")
}
/* Output:
1000 1000 1000.0
*/
```

Bir `Int`, -2^{31} ile $+2^{31}-1$ arasındaki değerleri tutar. Tamsayı değerleri taşabilir; örneğin, `Int.MAX_VALUE`'a herhangi bir şey eklemek bir taşma oluşturur:

```
// Summary1/Overflow.kt

fun main() {
  println(Int.MAX_VALUE + 1)
  println(Int.MAX_VALUE + 1L)
}
/* Output:
-2147483648
2147483648
*/
```

İkinci `println()` ifadesinde, `L`'yi `1`'e ekleyerek tüm ifadeyi `Long` türüne dönüştürüyoruz, bu da taşmayı önlüyor. (Bir `Long`, -2^{63} ile $+2^{63}-1$ arasındaki değerleri tutabilir).

Bir `Int`'i başka bir `Int` ile böldüğünüzde, Kotlin bir `Int` sonucu üretir ve kalan kısmı keser. Dolayısıyla, `1/2` `0` üretir. Eğer bir `Double` işin içine girerse, `Int` işlemden önce *Double*'a terfi eder, bu yüzden `1.0/2` `0.5` üretir.

Aşağıdaki `d1`'in `3.4` üretmesini bekleyebilirsiniz:

```
// Summary1/Truncation.kt

fun main() {
  val d1: Double = 3.0 + 2 / 5
  println(d1)
  val d2: Double = 3 + 2.0 / 5
  println(d2)
}
/* Output:
3.0
3.4
*/
```

Değerlendirme sırası nedeniyle, öyle olmaz. Kotlin önce `2`'yi `5` ile böler ve tam sayı matematiği `0` üretir ve bu, `3.0` cevabını verir. Aynı değerlendirme sırası `d2` için beklenen sonucu *verir*. `2.0`'yi `5` ile bölmek `0.4` üretir. `3`, bir `Double` ile toplandığı için (`0.4`), `Double` olarak yükseltilir ve bu da `3.4` üretir.

Değerlendirme sırasını anlamak, bir programın ne yaptığını hem mantıksal işlemler (Boolean ifadeleri) hem de matematiksel işlemlerle deşifre etmenize yardımcı olur.

Değerlendirme sırası hakkında emin değilseniz, niyetinizi zorlamak için parantez kullanın. Bu aynı zamanda kodunuzu okuyanlara da netlik sağlar.

`while` ile Tekrar

Bir `while` döngüsü, kontrol eden *Boolean ifadesi* `true` ürettiği sürece devam eder:

```
while (Boolean-expression) {
  // Code to be repeated
}
```

Boolean ifadesi döngünün başında ve her bir yineleme öncesinde bir kez daha değerlendirilir.

```
// Summary1/While.kt

fun testCondition(i: Int) = i < 100

fun main() {
  var i = 0
  while (testCondition(i)) {
    print(".")
    i += 10
  }
}
/* Output:
..........
*/
```

Kotlin, `testCondition()` için sonuç türü olarak `Boolean` türünü çıkarır.

Atama operatörlerinin kısa versiyonları tüm matematiksel işlemler için mevcuttur (`+=`, `-=`, `*=`, `/=`, `%=`). Kotlin ayrıca, artırma ve azaltma operatörlerini `++` ve `--`, hem önek hem de sonek formunda destekler.

`while`, `do` anahtar kelimesi ile kullanılabilir:

```
do {
  // Code to be repeated
} while (Boolean-expression)
```

`While.kt` Yeniden yazılıyor:

```
// Summary1/DoWhile.kt

fun main() {
  var i = 0
  do {
    print(".")
    i += 10
  } while (testCondition(i))
}
/* Output:
..........
*/
```

`while` ve `do-while` arasındaki tek fark, `do-while` gövdesinin, Boolean ifadesi ilk seferinde `false` üretsin ya da üretmesin, en az bir kez çalışmasıdır.

Döngüler ve Aralıklar

Birçok programlama dili, yineleyici bir nesneye indekslemek için tamsayılar arasında geçiş yapar. Kotlin'in `for` döngüsü, aralıklar ve `String`ler gibi yineleyici nesnelerden doğrudan eleman almanıza olanak tanır. Örneğin, bu `for` döngüsü, `"Kotlin"` `String`indeki her karakteri seçer:

```
// Summary1/StringIteration.kt

fun main() {
  for (c in "Kotlin") {
    print("$c ")
    // c + = 1 // error:
    // val cannot be reassigned
  }
}
/* Output:
K o t l i n
*/
```

`c` ne `var` ne de `val` olarak açıkça tanımlanabilir—Kotlin otomatik olarak onu bir `val` yapar ve tipini `Char` olarak çıkarır (tipi açıkça belirtebilirsiniz, ancak pratikte bu nadiren yapılır).

Tamsayı değerlerinden geçmek için *aralıkları* kullanabilirsiniz:

```
// Summary1/RangeOfInt.kt

fun main() {
  for (i in 1..10) {
    print("$i ")
  }
}
/* Output:
1 2 3 4 5 6 7 8 9 10
*/
```

`..` ile bir aralık oluşturmak her iki sınırı da içerir, ancak `until` üst sınır noktasını hariç tutar: `1 until 10` ile `1..9` aynıdır. `step` kullanarak bir artış değeri belirtebilirsiniz: `1..21 step 3`.

`in` Anahtar Kelimesi

`for` döngü yinelemesi sağlayan aynı `in`, bir aralıkta üyelik kontrolü yapmanıza da olanak tanır. `!in` test edilen değer *aralıkta değilse* `true` döndürür:

```
// Summary1/Membership.kt

fun inNumRange(n: Int) = n in 50..100

fun notLowerCase(ch: Char) = ch !in 'a'..'z'

fun main() {
  val i1 = 11
  val i2 = 100
  val c1 = 'K'
  val c2 = 'k'
  println("$i1 ${inNumRange(i1)}")
  println("$i2 ${inNumRange(i2)}")
  println("$c1 ${notLowerCase(c1)}")
  println("$c2 ${notLowerCase(c2)}")
}
/* Output:
11 false
100 true
K true
k false
*/
```

`in` ayrıca kayan nokta aralıklarında üyelik testi yapmak için de kullanılabilir, ancak bu tür aralıklar yalnızca `..` kullanılarak tanımlanabilir ve `until` kullanılamaz.

İfadeler ve Deyimler

Çoğu programlama dilindeki en küçük kullanışlı kod parçası ya bir *deyim* ya da bir *ifade*dir. Bunların temel bir farkı vardır:

- *Bir deyim durumu değiştirir.*
- *Bir ifade ifade eder.*

Yani, bir ifade bir sonuç üretir, oysa bir deyim üretmez. Çünkü bir şey döndürmediği için, bir deyim, çevresinin durumunu değiştirmelidir (yani, bir *yan etki* yaratmalıdır) ki herhangi bir işe yarasın.

Kotlin'deki neredeyse her şey bir ifadedir:

```
val hours = 10
val minutesPerHour = 60
val minutes = hours * minutesPerHour
```

Her durumda, = işaretinin sağındaki her şey bir ifadedir ve sol taraftaki tanımlayıcıya atanan bir sonuç üretir.

`println()` gibi fonksiyonlar bir sonuç üretmiyor gibi görünse de, ifadeler olduğu için *bir şey* döndürmeleri gerekir. Kotlin bu tür durumlar için özel bir `Unit` türüne sahiptir:

```
// Summary1/UnitReturn.kt

fun main() {
  val result = println("returns Unit")
  println(result)
}
/* Output:
returns Unit
kotlin.Unit
*/
```

Deneyimli programcılar, bu atom için egzersizleri yaptıktan sonra Özet 2'ye gidebilirler.

Egzersizler ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Bölüm II: Nesnelere Giriş

> *Nesneler*, Kotlin de dahil olmak üzere birçok modern dilin temelini oluşturur.

Nesne yönelimli (OO) bir programlama dilinde, çözdüğünüz problemdeki "isimleri" keşfeder ve bu isimleri nesnelere çevirirsiniz. Nesneler veri tutar ve eylemler gerçekleştirir. Nesne yönelimli bir dil, nesneler oluşturur ve kullanır.

Kotlin sadece nesne yönelimli değildir; aynı zamanda *fonksiyonel*dir. Fonksiyonel diller, gerçekleştirilen eylemlere ("fiiller") odaklanır. Kotlin, hibrit bir nesne-fonksiyonel dildir.

- Bu bölüm, nesne yönelimli programlamanın temel prensiplerini açıklar.
- Bölüm IV: Fonksiyonel Programlama fonksiyonel programlamayı tanıtır.
- Bölüm V: Nesne Yönelimli Programlama nesne yönelimli programlamayı detaylı olarak ele alır.

Her Yerde Nesneler

> Nesneler, *özellik* (`val` ve `var`) kullanarak veri depolar ve bu verilerle fonksiyonlar kullanarak işlemler yapar.

Bazı tanımlar:

- *Sınıf*: Esasen yeni bir veri tipi olan özellikleri ve fonksiyonları tanımlar. Sınıflar, *kullanıcı tanımlı türler* olarak da adlandırılır.
- *Üye*: Bir sınıfın ya özelliği ya da fonksiyonudur.
- *Üye fonksiyon*: Yalnızca belirli bir sınıfın nesneleri ile çalışan bir fonksiyondur.
- *Nesne oluşturma*: Bir sınıfın `val` veya `var` türünde bir örneğini oluşturma işlemi. Bu işlem aynı zamanda o sınıfın *örneğini oluşturma* olarak da adlandırılır.

Sınıflar *durum* ve *davranış* tanımladığı için, `Double` veya `Boolean` gibi dahili türlerin örneklerine bile nesne olarak atıfta bulunabiliriz.

Kotlin'in `IntRange` sınıfını düşünün:

```
// ObjectsEverywhere/IntRanges.kt

fun main() {
  val r1 = IntRange(0, 10)
  val r2 = IntRange(5, 7)
  println(r1)
  println(r2)
}
/* Output:
0..10
5..7
*/
```

`IntRange` *sınıfının* iki nesnesini (örneğini) oluşturuyoruz. Her nesnenin bellekte kendi depolama alanı vardır. `IntRange` bir sınıftır, ancak 0'dan 10'a kadar olan belirli bir `r1` aralığı, `r2` aralığından farklı olan bir nesnedir.

`IntRange` nesnesi için birçok işlem mevcuttur. Bazıları `sum()` gibi basittir, bazıları ise kullanmadan önce daha fazla anlayış gerektirir. Argümanlara ihtiyaç duyan birini çağırmaya çalışırsanız, IDE bu argümanları sizden ister.

Belirli bir üye fonksiyon hakkında bilgi edinmek için Kotlin dokümantasyonunda[19] arayın. Sayfanın sağ üst köşesindeki büyüteç simgesine dikkat edin. Ona tıklayın ve arama kutusuna `IntRange` yazın. Çıkan arama sonuçlarından `kotlin.ranges > IntRange`'e tıklayın. `IntRange` sınıfının dokümantasyonunu göreceksiniz. Sınıfın tüm üye fonksiyonlarını—sınıfın *Uygulama Programlama Arayüzü* (API)—inceleyebilirsiniz. Şu anda çoğunu anlamayacak olsanız da, Kotlin dokümantasyonunda arama yapmaya alışmak faydalıdır.

`IntRange` bir tür nesnedir ve bir nesnenin tanımlayıcı özelliği üzerinde işlemler gerçekleştirilmesidir. "Bir işlem gerçekleştirmek" yerine, *bir üye fonksiyonu çağırmak* deriz. Bir nesne için bir üye fonksiyonu çağırmak için, nesne tanımlayıcısıyla başlayın, ardından bir nokta ve ardından işlemin adını yazın:

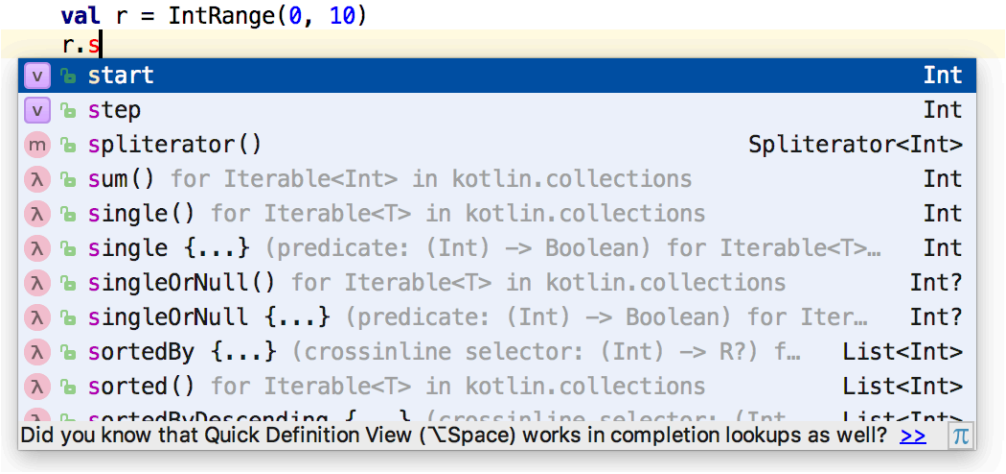
```
// ObjectsEverywhere/RangeSum.kt

fun main() {
  val r = IntRange(0, 10)
  println(r.sum())
}
/* Output:
55
*/
```

`sum()` `IntRange` için tanımlanmış bir üye fonksiyon olduğundan, `r.sum()` diyerek çağırırsınız. Bu, o `IntRange` içindeki tüm sayıları toplar.

Daha önceki nesne yönelimli diller, bir nesne için bir üye fonksiyonu çağırmayı tanımlamak için "mesaj göndermek" ifadesini kullanıyordu. Bazen bu terminolojiyi hâlâ görebilirsiniz.

Sınıflar birçok işleve (üye fonksiyonlar) sahip olabilir. Bir IDE (entegre geliştirme ortamı) kullanarak sınıfları keşfetmek kolaydır ve bu ortamlar *kod tamamlama* adı verilen bir özelliği içerir. Örneğin, IntelliJ IDEA içinde bir nesne tanımlayıcısının ardından `.s` yazarsanız, o nesnenin `s` ile başlayan tüm üyelerini gösterir:

[19] https://kotlinlang.org/api/latest/jvm/stdlib/index.html



Kod Tamamlama

Diğer nesneler üzerinde kod tamamlama kullanmayı deneyin. Örneğin, bir `String`'i ters çevirebilir veya tüm karakterleri küçük harfe dönüştürebilirsiniz:

```
// ObjectsEverywhere/Strings.kt

fun main() {
  val s = "AbcD"
  println(s.reversed())
  println(s.lowercase())
}
/* Output:
DcbA
abcd
*/
```

Bir `String`i kolayca bir tamsayıya ve tekrar geri dönüştürebilirsiniz:

```
// ObjectsEverywhere/Conversion.kt

fun main() {
  val s = "123"
  println(s.toInt())
  val i = 123
  println(i.toString())
}
/* Output:
123
123
*/
```

Kitabın ilerleyen bölümlerinde, dönüştürmek istediğiniz `String` doğru bir tam sayı değeri temsil etmediğinde karşılaşabileceğiniz durumlarla başa çıkma stratejilerini tartışacağız.

Bir sayısal türden başka bir sayısal türe de dönüştürebilirsiniz. Karışıklığı önlemek için, sayısal türler arasındaki dönüşümler açıkça belirtilmiştir. Örneğin, bir `Int` `i`'yi `i.toLong()` çağırarak bir `Long`'a veya `i.toDouble()` çağırarak bir `Double`'a dönüştürebilirsiniz:

```
// ObjectsEverywhere/NumberConversions.kt

fun fraction(numerator: Long, denom: Long) =
  numerator.toDouble() / denom

fun main() {
  val num = 1
  val den = 2
  val f = fraction(num.toLong(), den.toLong())
  println(f)
}
/* Output:
0.5
*/
```

İyi tanımlanmış sınıflar, bir programcının anlamasını kolaylaştırır ve okuması kolay kod üretir.

Egzersizler ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Sınıflar Oluşturma

> `IntRange` ve `String` gibi önceden tanımlanmış türleri kullanabileceğiniz gibi, kendi nesne türlerinizi de oluşturabilirsiniz.

Gerçekten de, yeni türler oluşturmak, nesne yönelimli programlamada yapılan faaliyetlerin büyük bir kısmını oluşturur. Yeni türler oluşturmak için *sınıflar* tanımlarsınız.

Bir nesne, çözmeye çalıştığınız bir problemin çözümünün bir parçasıdır. Nesneleri kavramları ifade ederken düşünerek başlayın. İlk yaklaşıma göre, probleminizde bir "şey" keşfettiğinizde, bu şeyi çözümünüzde bir nesne olarak temsil edin.

Diyelim ki bir hayvanat bahçesindeki hayvanları yönetmek için bir program oluşturmak istiyorsunuz. Hayvanların nasıl davrandıklarına, ihtiyaçlarına, hangi hayvanlarla anlaştıklarına ve hangileriyle kavga ettiklerine göre farklı türleri kategorize etmek mantıklıdır. Bir hayvan türüyle ilgili farklı olan her şey, o hayvanın nesnesinin sınıflandırılmasında yakalanır. Kotlin, yeni bir nesne türü oluşturmak için `class` anahtar kelimesini kullanır:

```
// CreatingClasses/Animals.kt

// Create some classes:
class Giraffe
class Bear
class Hippo

fun main() {
  // Create some objects:
  val g1 = Giraffe()
  val g2 = Giraffe()
  val b = Bear()
  val h = Hippo()

  // Each object() is unique:
  println(g1)
```

```
  println(g2)
  println(h)
  println(b)
}
/* Sample output:
Giraffe@28d93b30
Giraffe@1b6d3586
Hippo@4554617c
Bear@74a14482
*/
```

Bir sınıf tanımlamak için `class` anahtar kelimesiyle başlayın, ardından yeni sınıfınız için bir tanımlayıcı ekleyin. Sınıf adı bir harf (A-Z, büyük veya küçük harf) ile başlamalıdır, ancak sayılar ve alt çizgiler gibi şeyleri içerebilir. Konvansiyona göre, bir sınıf adının ilk harfini büyük, tüm `val` ve `var`'ların ilk harfini küçük yaparız.

`Animals.kt`, üç yeni sınıf tanımlayarak başlar, ardından bu sınıfların dört nesnesini (aynı zamanda *örnek* olarak da adlandırılır) oluşturur.

`Giraffe` bir sınıftır, ancak Botswana'da yaşayan beş yaşında bir erkek zürafa bir *nesne*dir. Her nesne diğerlerinden farklıdır, bu yüzden onlara `g1` ve `g2` gibi isimler veriyoruz.

Son dört satırın oldukça şifreli çıktısına dikkat edin. `@` işaretinin önündeki kısım sınıf adıdır ve `@` işaretinden sonraki numara, nesnenin bilgisayarınızın belleğinde bulunduğu adresidir. Evet, bu harfler içermesine rağmen bir numaradır—buna "onaltılık notasyon"[20] denir. Programınızdaki her nesnenin kendine özgü bir adresi vardır.

Burada tanımlanan sınıflar (`Giraffe`, `Bear` ve `Hippo`) mümkün olduğunca basittir: tüm sınıf tanımı tek satırdır. Daha karmaşık sınıflar, bu sınıfın özelliklerini ve davranışlarını içeren bir *sınıf gövdesi* oluşturmak için süslü parantezler (`{` ve `}`) kullanır.

Bir sınıf içinde tanımlanan bir fonksiyon o sınıfa aittir. Kotlin'de bunlara sınıfın *üye fonksiyonları* denir. Java gibi bazı nesne yönelimli diller bunlara *yöntem* demeyi tercih eder, bu terim eski nesne yönelimli dillerden Smalltalk'tan gelmiştir. Kotlin'in fonksiyonel doğasını vurgulamak için, tasarımcılar *yöntem* terimini bırakmayı tercih ettiler, çünkü bazı yeni başlayanlar bu ayrımı kafa karıştırıcı buluyordu. Bunun yerine, dil boyunca *fonksiyon* terimi kullanılır.

[20] https://en.wikipedia.org/wiki/Hexadecimal

Belirsiz değilse, sadece "fonksiyon" diyeceğiz. Ayrımı yapmamız gerekirse:

- *Üye* fonksiyonlar bir sınıfa aittir.
- *Üst düzey* fonksiyonlar kendileri var olur ve bir sınıfın parçası değildir.

Burada, `bark()` `Dog` sınıfına aittir:

```
// CreatingClasses/Dog.kt

class Dog {
  fun bark() = "yip!"
}

fun main() {
  val dog = Dog()
}
```

`main()` fonksiyonunda bir `Dog` nesnesi oluşturup `val dog`'a atıyoruz. Kotlin, `dog`'u hiç kullanmadığımız için bir uyarı veriyor.

Üye fonksiyonlar, nesne adı, ardından bir `.` (nokta), ardından fonksiyon adı ve parametre listesi ile çağrılır (*invoked*). Burada, `meow()` fonksiyonunu çağırıp sonucu görüntülüyoruz:

```
// CreatingClasses/Cat.kt

class Cat {
  fun meow() = "mrrrow!"
}

fun main() {
  val cat = Cat()
  // Call 'meow()' for 'cat':
  val m1 = cat.meow()
  println(m1)
}
/* Output:
mrrrow!
*/
```

Bir üye fonksiyon, belirli bir sınıf örneği üzerinde çalışır. `meow()` çağrısı yaptığınızda, bunu bir nesne ile çağırmalısınız. Çağrı sırasında, `meow()` o nesnenin diğer üyelerine erişebilir.

Bir üye fonksiyonu çağırırken, Kotlin ilgilenilen nesneyi sessizce o nesneye bir referans geçirerek takip eder. Bu referans, üye fonksiyon içinde `this` anahtar kelimesi kullanılarak kullanılabilir.

Üye fonksiyonlar, sadece bu öğelerin adlarını belirterek sınıf içindeki diğer öğelere özel erişim sağlar. Ayrıca, bu öğelere erişimi `this` kullanarak açıkça *nitelendirebilirsiniz*. Burada, `exercise()` `speak()` fonksiyonunu nitelikli ve niteliksiz olarak çağırır:

```
// CreatingClasses/Hamster.kt

class Hamster {
  fun speak() = "Squeak! "
  fun exercise() =
    this.speak() +   // Qualified with 'this'
      speak() +      // Without 'this'
      "Running on wheel"
}

fun main() {
  val hamster = Hamster()
  println(hamster.exercise())
}
/* Output:
Squeak! Squeak! Running on wheel
*/
```

`exercise()` içinde, önce açık bir `this` ile `speak()` çağırıyoruz ve sonra bu nitelemeyi atlıyoruz.

Bazen gereksiz bir açık `this` içeren kodlar göreceksiniz. Bu tür kodlar genellikle `this`'in ya gerekli olduğu ya da tarzın bir parçası olduğu farklı bir dili bilen programcılardan gelir. Gereksiz bir özelliği kullanmak, okuyucunun neden bunu yaptığınızı anlamaya çalışırken zaman harcamasına neden olur. Gereksiz `this` kullanımından kaçınmanızı öneririz.

Sınıfın dışında, `hamster.exercise()` ve `hamster.speak()` demeniz gerekir.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Özellikler

> Bir *özellik*, bir sınıfın parçası olan bir `var` veya `val`'dir.

Bir özellik tanımlamak, bir sınıf içinde *durumu korur*. Durumu korumak, sadece bir veya daha fazla bağımsız fonksiyon yazmak yerine bir sınıf yaratmanın ana motivasyon nedenidir.

Bir `var` özelliği yeniden atanabilirken, bir `val` özelliği atanamaz. Her nesne, özellikler için kendi depolamasını alır:

```
// Properties/Cup.kt

class Cup {
  var percentFull = 0
}

fun main() {
  val c1 = Cup()
  c1.percentFull = 50
  val c2 = Cup()
  c2.percentFull = 100

  println(c1.percentFull)
  println(c2.percentFull)
}
/* Output:
50
100
*/
```

Bir sınıfın içinde `var` veya `val` tanımlamak, bir fonksiyon içinde tanımlamak gibidir. Ancak, `var` veya `val` o sınıfın *parçası* haline gelir ve ona *nokta gösterimi* kullanarak, nesne ile özellik adı arasına bir nokta koyarak başvurmanız gerekir. `percentFull` referanslarının her birinde nokta gösteriminin kullanıldığını görebilirsiniz.

`percentFull` özelliği, ilgili `Cup` nesnesinin durumunu temsil eder. `c1.percentFull` ve `c2.percentFull` farklı değerler içerir, bu da her nesnenin kendi depolamasının olduğunu gösterir.

Bir üye fonksiyon kendi nesnesi içindeki bir özelliğe nokta gösterimi kullanmadan başvurabilir (yani, *nitelendirmeden*):

```
// Properties/Cup2.kt

class Cup2 {
  var percentFull = 0
  val max = 100
  fun add(increase: Int): Int {
    percentFull += increase
    if (percentFull > max)
      percentFull = max
    return percentFull
  }
}

fun main() {
  val cup = Cup2()
  cup.add(50)
  println(cup.percentFull)
  cup.add(70)
  println(cup.percentFull)
}
/* Output:
50
100
*/
```

`add()` üye fonksiyonu, `percentFull` değerine `artış` eklemeye çalışır fakat bu değerin %100'ü geçmemesini sağlar.

Bir sınıfın dışından hem özelliklere hem de üye fonksiyonlara nitelik kazandırmalısınız.

Üst düzey özellikler tanımlayabilirsiniz:

```
// Properties/TopLevelProperty.kt

val constant = 42

var counter = 0

fun inc() {
  counter++
}
```

Üst düzey bir `val` tanımlamak güvenlidir çünkü değiştirilemez. Ancak, değiştirilebilir (`var`) üst düzey bir özellik tanımlamak bir *anti-kalıp* olarak kabul edilir. Programınız karmaşıklaştıkça, *paylaşılan değişken durumu* doğru bir şekilde anlamak zorlaşır. Kod tabanınızdaki herkes `var counter`a erişebiliyorsa, doğru şekilde değişeceğini garanti edemezsiniz: `inc()` `counter`ı bir artırırken, programın başka bir bölümü `counter`ı on azaltabilir ve belirsiz hatalara yol açabilir. Değişken durumu bir sınıf içinde korumak en iyisidir. Görünürlüğü Sınırlandırma bölümünde onu gerçekten gizli yapmayı göreceksiniz.

`var`ların değiştirilebilirken `val`ların değiştirilemez olduğunu söylemek bir basitleştirmedir. Bir benzetme olarak, bir `evi val` olarak ve `ev` içindeki bir `kanepeyi var` olarak düşünün. `Kanepeyi` değiştirebilirsiniz çünkü o bir `var`. Ancak `evi` yeniden atayamazsınız çünkü o bir `val`dir:

```
// Properties/ChangingAVal.kt

class House {
  var sofa: String = ""
}

fun main() {
  val house = House()
  house.sofa = "Simple sleeper sofa: $89.00"
  println(house.sofa)
  house.sofa = "New leather sofa: $3,099.00"
  println(house.sofa)
  // Cannot reassign the val to a new House:
  // house = House()
}
/* Output:
Simple sleeper sofa: $89.00
```

```
New leather sofa: $3,099.00
*/
```

Her ne kadar house bir `val` olsa da, `class House` içinde `sofa` bir `var` olduğu için nesnesi değiştirilebilir. house'u bir `val` olarak tanımlamak, sadece yeni bir nesneye yeniden atanmasını engeller.

Bir özelliği `val` yaparsak, yeniden atanamaz:

```
// Properties/AnUnchangingVar.kt

class Sofa {
  val cover: String = "Loveseat cover"
}

fun main() {
  var sofa = Sofa()
  // Not allowed:
  // sofa.cover = "New cover"
  // Reassigning a var:
  sofa = Sofa()
}
```

Her ne kadar `sofa` bir `var` olsa da, `class Sofa` içindeki `cover` bir `val` olduğundan objesi değiştirilemez. Ancak, `sofa` yeni bir objeye yeniden atanabilir.

house ve `sofa` gibi tanımlayıcılar hakkında sanki objelermiş gibi konuştuk. Aslında onlar objelere olan *referanslar*dır. Bunu görmenin bir yolu, iki tanımlayıcının aynı objeye referans verebildiğini gözlemlemektir:

```
// Properties/References.kt

class Kitchen {
  var table: String = "Round table"
}

fun main() {
  val kitchen1 = Kitchen()
  val kitchen2 = kitchen1
  println("kitchen1: ${kitchen1.table}")
  println("kitchen2: ${kitchen2.table}")
  kitchen1.table = "Square table"
```

```
  println("kitchen1: ${kitchen1.table}")
  println("kitchen2: ${kitchen2.table}")
}
/* Output:
kitchen1: Round table
kitchen2: Round table
kitchen1: Square table
kitchen2: Square table
*/
```

`kitchen1 table` üzerinde değişiklik yaptığında, `kitchen2` bu değişikliği görür. `kitchen1.table` ve `kitchen2.table` aynı çıktıyı gösterir.

`var` ve `val`'ın nesneler yerine referansları kontrol ettiğini unutmayın. Bir `var`, bir referansı farklı bir nesneye yeniden bağlamanıza izin verirken, bir `val` bunu yapmanızı engeller.

Değişkenlik, bir nesnenin durumunu değiştirebileceği anlamına gelir. Yukarıdaki örneklerde, `class House` ve `class Kitchen` değişken nesneler tanımlarken, `class Sofa` değişmez nesneler tanımlar.

Egzersizler ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Yapıcılar

> Yeni bir nesneyi bir *yapıcıya* bilgi geçirerek başlatırsınız.

Her nesne izole bir dünyadır. Bir program, nesneler topluluğudur, bu yüzden her bir nesnenin doğru başlatılması, başlatma probleminin büyük bir kısmını çözer. Kotlin, doğru nesne başlatmayı garanti eden mekanizmalar içerir.

Bir yapıcı, yeni bir nesneyi başlatan özel bir üye fonksiyon gibidir. Bir yapıcının en basit formu, tek satırlık bir sınıf tanımıdır:

```
// Constructors/Wombat.kt

class Wombat

fun main() {
  val wombat = Wombat()
}
```

`main()` içinde `Wombat()` çağrısı bir `Wombat` nesnesi oluşturur. Eğer başka bir nesne yönelimli dilden geliyorsanız burada bir `new` anahtar kelimesi görmeyi bekleyebilirsiniz, ancak `new` Kotlin'de gereksiz olduğu için kullanılmamıştır.

Bir yapıcıya bilgi aktarmak için bir parametre listesi kullanırsınız, tıpkı bir fonksiyonda olduğu gibi. Burada, `Alien` yapıcısı tek bir argüman alır:

```
// Constructors/Arg.kt

class Alien(name: String) {
  val greeting = "Poor $name!"
}

fun main() {
  val alien = Alien("Mr. Meeseeks")
  println(alien.greeting)
  // alien.name // Error     // [1]
}
/* Output:
Poor Mr. Meeseeks!
*/
```

Bir `Alien` nesnesi oluşturmak, bir argüman gerektirir (argümansız olarak deneyin). `name`, yapıcı içinde `greeting` özelliğini başlatır, ancak yapıcı dışında erişilemez—satır **[1]**'i yorumsuz hale getirmeyi deneyin.

Yapıcı parametresinin sınıf gövdesi dışında erişilebilir olmasını istiyorsanız, parametre listesindeki `var` veya `val` olarak tanımlayın:

```
// Constructors/VisibleArgs.kt

class MutableNameAlien(var name: String)

class FixedNameAlien(val name: String)

fun main() {
  val alien1 =
    MutableNameAlien("Reverse Giraffe")
  val alien2 =
    FixedNameAlien("Krombopulos Michael")

  alien1.name = "Parasite"
  // Can't do this:
  // alien2.name = "Parasite"
}
```

Bu sınıf tanımlamalarının açıkça belirtilmiş sınıf gövdeleri yoktur—gövdeler örtük olarak varsayılır.

`name` bir `var` veya `val` olarak tanımlandığında, bir özellik haline gelir ve dolayısıyla yapıcı dışında erişilebilir olur. `val` yapıcı parametreleri değiştirilemezken, `var` yapıcı parametreleri değiştirilebilir.

Sınıfınızın birçok yapıcı parametresi olabilir:

```
// Constructors/MultipleArgs.kt

class AlienSpecies(
  val name: String,
  val eyes: Int,
  val hands: Int,
  val legs: Int
) {
  fun describe() =
    "$name with $eyes eyes, " +
      "$hands hands and $legs legs"
}

fun main() {
  val kevin =
    AlienSpecies("Zigerion", 2, 2, 2)
  val mortyJr =
    AlienSpecies("Gazorpian", 2, 6, 2)
  println(kevin.describe())
  println(mortyJr.describe())
}
/* Output:
Zigerion with 2 eyes, 2 hands and 2 legs
Gazorpian with 2 eyes, 6 hands and 2 legs
*/
```

Karmaşık Yapıcılar bölümünde, yapıcıların karmaşık başlatma mantığı içerebileceğini göreceksiniz.

Bir nesne `String` beklenen bir yerde kullanıldığında, Kotlin nesnenin `toString()` üye işlevini çağırır. Eğer bir tane yazmazsanız, yine de varsayılan bir `toString()` alırsınız:

```
// Constructors/DisplayAlienSpecies.kt

fun main() {
  val krombopulosMichael =
    AlienSpecies("Gromflomite", 2, 2, 2)
  println(krombopulosMichael)
}
/* Sample output:
AlienSpecies@4d7e1886
*/
```

Varsayılan `toString()` çok kullanışlı değildir—sınıf adını ve nesnenin fiziksel adresini üretir (bu, her program çalıştırıldığında değişir). Kendi `toString()` yönteminizi tanımlayabilirsiniz:

```
// Constructors/Scientist.kt

class Scientist(val name: String) {
  override fun toString() =
    "Scientist('$name')"
}

fun main() {
  val zeep = Scientist("Zeep Xanflorp")
  println(zeep)
}
/* Output:
Scientist('Zeep Xanflorp')
*/
```

`override` bizim için yeni bir anahtar kelime. Burada gereklidir çünkü `toString()` zaten bir tanıma sahiptir ve bu tanım ilkel sonucu üretir. `override`, Kotlin'e evet, varsayılan `toString()` tanımını kendi tanımımızla değiştirmek istediğimizi belirtir. `override`'nin açıklığı kodu netleştirir ve hataları önler.

Bir nesnenin içeriğini uygun bir şekilde gösteren bir `toString()`, programlama hatalarını bulmak ve düzeltmek için faydalıdır. *Hata ayıklama* sürecini basitleştirmek için IDE'ler, bir programın yürütülmesindeki her adımı gözlemlemenize ve nesnelerinizin içine bakmanıza olanak tanıyan *hata ayıklayıcılar*[21] sağlar.

Egzersizler ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

[21] https://www.jetbrains.com/help/idea/debugging-code.html

Görünürlüğü Kısıtlamak

> Bir parça kodu birkaç gün veya hafta boyunca bırakıp sonra geri döndüğünüzde, onu yazmanın çok daha iyi bir yolunu görebilirsiniz.

Bu, çalışan kodu daha okunabilir, anlaşılabilir ve dolayısıyla sürdürülebilir hale getirmek için yeniden yazan *refactoring*in ana motivasyonlarından biridir.

Kodunuzu değiştirme ve iyileştirme isteğinizde bir gerilim vardır. Tüketiciler (*istemci programcılar*) kodunuzun sabit kalmasını gerektirir. Siz onu değiştirmek istersiniz, onlar ise aynı kalmasını isterler.

Bu özellikle kütüphaneler için önemlidir. Bir kütüphanenin tüketicileri, o kütüphanenin yeni bir sürümü için kodu yeniden yazmak istemezler. Ancak, kütüphane yaratıcısının değişiklikler ve iyileştirmeler yapma özgürlüğüne sahip olması ve istemci kodunun bu değişikliklerden etkilenmeyeceğinden emin olması gerekir.

Dolayısıyla, yazılım tasarımında ana bir düşünce şudur:

> *Değişen şeyleri sabit kalan şeylerden ayırın.*

Görünürlüğü kontrol etmek için Kotlin ve diğer bazı diller *access modifiers* sağlar. Kütüphane yaratıcıları `public`, `private`, `protected` ve `internal` modifikasyonlarını kullanarak istemci programcı tarafından neyin erişilebilir olup olmadığını belirlerler. Bu bölüm `public` ve `private` konularını kapsar, `internale` kısa bir giriş yapar. `protected`ı kitabın ilerleyen bölümlerinde açıklarız.

`private` gibi bir access modifier bir sınıf, fonksiyon veya özellik tanımından önce görünür. Bir access modifier sadece o belirli tanım için erişimi kontrol eder.

Bir `public` tanım istemci programcılar tarafından erişilebilir, bu yüzden o tanımdaki değişiklikler istemci kodunu doğrudan etkiler. Bir modifikasyon sağlamazsanız, tanımınız otomatik olarak `public` olur, bu yüzden `public` teknik olarak gereksizdir. Ancak, bazen netlik açısından yine de `public` olarak belirtebilirsiniz.

Bir `private` tanım gizlidir ve yalnızca aynı sınıfın diğer üyeleri tarafından erişilebilir. Bir `private` tanımı değiştirmek veya hatta kaldırmak istemci programcıları doğrudan etkilemez.

`private` sınıflar, üst seviye fonksiyonlar ve üst seviye özellikler yalnızca o dosya içinde erişilebilir:

```
// Visibility/RecordAnimals.kt

private var index = 0                   // [1]

private class Animal(val name: String) // [2]

private fun recordAnimal(               // [3]
  animal: Animal
) {
  println("Animal #$index: ${animal.name}")
  index++
}

fun recordAnimals() {
  recordAnimal(Animal("Tiger"))
  recordAnimal(Animal("Antelope"))
}

fun recordAnimalsCount() {
  println("$index animals are here!")
}
```

`private` üst düzey özelliklere (**[1]**), sınıflara (**[2]**) ve fonksiyonlara (**[3]**) `RecordAnimals.kt` dosyasındaki diğer fonksiyonlar ve sınıflardan erişebilirsiniz. Kotlin, başka bir dosya içinden `private` üst düzey bir elemana erişmenizi engelleyerek, bunun dosyada `private` olduğunu size bildirir:

```
// Visibility/ObserveAnimals.kt

fun main() {
  // Can't access private members
  // declared in another file.
  // Class is private:
  // val rabbit = Animal("Rabbit")
  // Function is private:
  // recordAnimal(rabbit)
  // Property is private:
  // index++

  recordAnimals()
  recordAnimalsCount()
}
/* Output:
Animal #0: Tiger
Animal #1: Antelope
2 animals are here!
*/
```

Gizlilik en yaygın olarak bir sınıfın üyeleri için kullanılır:

```
// Visibility/Cookie.kt

class Cookie(
  private var isReady: Boolean  // [1]
) {
  private fun crumble() =       // [2]
    println("crumble")

  public fun bite() =           // [3]
    println("bite")

  fun eat() {                   // [4]
    isReady = true              // [5]
    crumble()
    bite()
  }
}

fun main() {
```

```
  val x = Cookie(false)
  x.bite()
  // Can't access private members:
  // x.isReady
  // x.crumble()
  x.eat()
}
/* Output:
bite
crumble
bite
*/
```

- **[1]** İçerdiği sınıfın dışından erişilemeyen bir `private` özellik.
- **[2]** Bir `private` üye fonksiyon.
- **[3]** Herkesin erişebileceği bir `public` üye fonksiyon.
- **[4]** Erişim belirleyicisi olmayanlar `public` anlamına gelir.
- **[5]** Aynı sınıfın üyeleri dışında kimse `private` üyelere erişemez.

`private` anahtar kelimesi, o üye dışında kimsenin o üyeye erişemeyeceği anlamına gelir. Diğer sınıflar `private` üyelere erişemez, bu nedenle kendinizi ve iş arkadaşlarınızı da sınıfa karşı yalıtıyormuşsunuz gibi olur. `private` ile, aynı paketteki başka bir sınıfı etkileyip etkilemeyeceği konusunda endişelenmeden o üyeyi özgürce değiştirebilirsiniz. Bir kütüphane tasarımcısı olarak, genellikle her şeyi mümkün olduğunca `private` tutar ve sadece fonksiyonları ve sınıfları kullanıcı programcılara açarsınız.

Bir sınıf için *yardımcı fonksiyon* olan herhangi bir üye fonksiyon `private` yapılabilir, böylece paketin başka yerinde yanlışlıkla kullanmaz ve bu fonksiyonu değiştirme veya kaldırma hakkınızı kendinizden almazsınız.

Bir sınıf içindeki bir `private` özellik için de aynı şey geçerlidir. Temel uygulamayı açığa çıkarmanız gerekmediği sürece (sandığınızdan daha az olasıdır), özellikleri `private` yapın. Ancak, bir sınıf içindeki bir nesneye ait bir referansın `private` olması, başka bir nesnenin aynı nesneye ait bir `public` referansına sahip olamayacağı anlamına gelmez:

```
// Visibility/MultipleRef.kt

class Counter(var start: Int) {
  fun increment() {
    start += 1
  }
  override fun toString() = start.toString()
}

class CounterHolder(counter: Counter) {
  private val ctr = counter
  override fun toString() =
    "CounterHolder: " + ctr
}

fun main() {
  val c = Counter(11)                  // [1]
  val ch = CounterHolder(c)            // [2]
  println(ch)
  c.increment()                        // [3]
  println(ch)
  val ch2 = CounterHolder(Counter(9)) // [4]
  println(ch2)
}
/* Output:
CounterHolder: 11
CounterHolder: 12
CounterHolder: 9
*/
```

- **[1]** `c` artık aşağıdaki satırda `CounterHolder` nesnesinin oluşturulmasıyla *çevrili* kapsamda tanımlanmıştır.
- **[2]** `c`'yi `CounterHolder` yapıcısına argüman olarak geçirmek, yeni `CounterHolder`'ın artık `c`'nin referans verdiği aynı `Counter` nesnesine referans verdiği anlamına gelir.
- **[3]** `ch` içinde `özel` olması gereken `Counter`, hala `c` aracılığıyla manipüle edilebilir.
- **[4]** `Counter(9)`'un `CounterHolder` dışında başka referansı yoktur, bu yüzden `ch2` dışında hiçbir şey tarafından erişilemez veya değiştirilemez.

Tek bir nesneye birden fazla referans bulundurmak *takma ad kullanımı* olarak adlandırılır ve şaşırtıcı davranışlara neden olabilir.

Modüller

Bu kitaptaki küçük örneklerin aksine, gerçek programlar genellikle büyüktür. Bu tür programları bir veya daha fazla *modüle* bölmek faydalı olabilir. Bir modül, kod tabanının mantıksal olarak bağımsız bir parçasıdır. Bir projeyi modüllere bölme şekliniz, Gradle[22] veya Maven[23] gibi derleme sistemine bağlıdır ve bu kitabın kapsamı dışındadır.

Bir `dahili` tanım, yalnızca tanımlandığı modül içinde erişilebilir. `dahili`, `özel` ve `genel` arasında bir yerde yer alır—`özel` çok kısıtlayıcı olduğunda ancak bir öğenin `genel` API'nin bir parçası olmasını istemediğinizde kullanın. Kitapta örneklerde veya alıştırmalarda `dahili` kullanmıyoruz.

Modüller daha yüksek seviyeli bir kavramdır. Aşağıdaki atom *paketleri* tanıtır, bu da daha ince ayrıntılı yapılandırmayı mümkün kılar. Bir kütüphane genellikle birden fazla paketten oluşan tek bir modüldür, bu yüzden `dahili` öğeler kütüphane içinde kullanılabilir, ancak bu kütüphanenin tüketicileri tarafından erişilemez.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

[22] https://gradle.org/
[23] https://maven.apache.org/

Paketler

Programlamada temel bir prensip DRY kısaltmasıdır: *Tekrarlama Kendini*.

Birden fazla aynı kod parçası, düzeltmeler veya iyileştirmeler yaptığınızda bakım gerektirir. Bu yüzden kodu çoğaltmak sadece ekstra iş değildir—her çoğaltma hata yapma olasılıkları yaratır.

`import` anahtar kelimesi, diğer dosyalardan kodu yeniden kullanır. `import` kullanmanın bir yolu, bir sınıf, fonksiyon veya özellik adını belirtmektir:

```
import packagename.ClassName
import packagename.functionName
import packagename.propertyName
```

Bir *paket*, birbiriyle ilişkili kod koleksiyonudur. Her paket genellikle belirli bir problemi çözmek için tasarlanmıştır ve çoğu zaman birden fazla fonksiyon ve sınıf içerir. Örneğin, `kotlin.math` kütüphanesinden matematiksel sabitler ve fonksiyonlar ithal edebiliriz:

```
// Packages/ImportClass.kt
import kotlin.math.PI
import kotlin.math.cos  // Cosine

fun main() {
  println(PI)
  println(cos(PI))
  println(cos(2 * PI))
}
/* Output:
3.141592653589793
-1.0
1.0
*/
```

Bazen aynı ada sahip sınıflar veya fonksiyonlar içeren birden fazla üçüncü parti kütüphane kullanmak isteyebilirsiniz. `as` anahtar kelimesi, içe aktarma sırasında isimleri değiştirmenize olanak tanır:

```
// Packages/ImportNameChange.kt
import kotlin.math.PI as circleRatio
import kotlin.math.cos as cosine

fun main() {
  println(circleRatio)
  println(cosine(circleRatio))
  println(cosine(2 * circleRatio))
}
/* Output:
3.141592653589793
-1.0
1.0
*/
```

`as` is useful if a library name is poorly chosen or excessively long.

Bir kütüphane ismi kötü seçilmişse veya aşırı uzun ise `as` kullanışlıdır.

You can fully qualify an import in the body of your code. In the following example, the code might be less readable due to the explicit package names, but the origin of each element is absolutely clear:

Kodunuzun içinde bir içe aktarmayı tamamen nitelendirebilirsiniz. Aşağıdaki örnekte, belirgin paket isimleri nedeniyle kodun okunabilirliği azalabilir, ancak her bir öğenin kaynağı kesinlikle açıktır:

```
// Packages/FullyQualify.kt

fun main() {
  println(kotlin.math.PI)
  println(kotlin.math.cos(kotlin.math.PI))
  println(kotlin.math.cos(2 * kotlin.math.PI))
}
/* Output:
3.141592653589793
-1.0
1.0
*/
```

Bir paketten her şeyi içe aktarmak için yıldız (*) kullanın:

```
// Packages/ImportEverything.kt
import kotlin.math.*

fun main() {
  println(E)
  println(E.roundToInt())
  println(E.toInt())
}
/* Output:
2.718281828459045
3
2
*/
```

`kotlin.math` paketi, `Double` değerini en yakın tam sayıya yuvarlayan kullanışlı bir `roundToInt()` içerir, bu `toInt()`'den farklı olarak ondalık noktadan sonraki herhangi bir şeyi basitçe keser.

Kodunuzu yeniden kullanmak için `package` anahtar kelimesini kullanarak bir paket oluşturun. `package` ifadesi dosyadaki ilk yorum olmayan ifade olmalıdır. `package` ifadesinden sonra gelen paket adı, geleneksel olarak tamamen küçük harf olmalıdır:

```
// Packages/PythagoreanTheorem.kt
package pythagorean
import kotlin.math.sqrt

class RightTriangle(
  val a: Double,
  val b: Double
) {
  fun hypotenuse() = sqrt(a * a + b * b)
  fun area() = a * b / 2
}
```

Kaynak kod dosyasına, sınıf adının dosya adıyla aynı olmasını gerektiren Java'nın aksine, istediğiniz herhangi bir adı verebilirsiniz.

Kotlin, paketiniz için herhangi bir ad seçmenize izin verir, ancak paket adının paket dosyalarının bulunduğu dizin adıyla aynı olması iyi bir stil olarak kabul edilir (bu kitapta yer alan örneklerde her zaman böyle olmayacaktır).

`pythagorean` paketindeki öğeler artık `import` kullanılarak kullanılabilir:

```
// Packages/ImportPythagorean.kt
import pythagorean.RightTriangle

fun main() {
  val rt = RightTriangle(3.0, 4.0)
  println(rt.hypotenuse())
  println(rt.area())
}
/* Output:
5.0
6.0
*/
```

Bu kitabın geri kalanında, diğer dosyalarla isim çakışmalarını önlemek için `main()` dışında fonksiyonlar, sınıflar vb. tanımlayan herhangi bir dosya için `package` ifadelerini kullanacağız, ancak *sadece* `main()` içeren bir dosyaya genellikle bir `package` ifadesi koymayacağız.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Testing

> Sürekli test etme, hızlı program geliştirme için gereklidir.

Kodunuzun bir kısmını değiştirdiğinizde diğer kodları bozarsa, testleriniz sorunu hemen ortaya çıkarır. Sorunu hemen bulamazsanız, değişiklikler birikir ve hangi değişikliğin soruna neden olduğunu artık söyleyemezsiniz. Sorunu bulmak için *çok* daha fazla zaman harcarsınız.

Test etme kritik bir uygulamadır, bu yüzden kitabın başında tanıtıyoruz ve kitabın geri kalanında kullanıyoruz. Bu şekilde, test etmeyi programlama sürecinin standart bir parçası olarak benimsemiş olursunuz.

Kod doğruluğunu doğrulamak için `println()` kullanmak zayıf bir yaklaşımdır—her seferinde çıktıyı dikkatle incelemeli ve doğru olduğundan emin olmalısınız.

Bu kitabı kullanırken deneyiminizi basitleştirmek için kendi küçük test sistemimizi oluşturduk. Amaç, minimal bir yaklaşımdır:

1. İfadelerin beklenen sonucunu gösterir.
2. Tüm testler başarılı olduğunda bile programın çalıştığını bilmenizi sağlayacak çıktı sağlar.
3. Pratikte erken test etme kavramını kazandırır.

Kitap için faydalı olsa da, bizimkisi işyeri için bir test sistemi *değildir*. Diğerleri, böyle test sistemleri oluşturmak için uzun ve zorlu çalışmalar yapmıştır. Örneğin:

- JUnit[24] en popüler Java test çerçevelerinden biridir ve Kotlin içinde kolayca kullanılabilir.
- Kotest[25] özellikle Kotlin için tasarlanmıştır ve Kotlin dil özelliklerinden yararlanır.
- Spek Framework[26] *Spesifikasyon Testi* olarak adlandırılan farklı bir test türü üretir.

Test framework'ümüzü kullanmak için önce `import` etmeliyiz. Framework'ün temel öğeleri `eq` (*eşittir*) ve `neq` (*eşit değildir*)'dir:

[24] https://junit.org
[25] https://github.com/kotest/kotest
[26] https://spekframework.org/

```
// Testing/TestingExample.kt
import atomictest.*

fun main() {
  val v1 = 11
  val v2 = "Ontology"

  // 'eq' means "equals":
  v1 eq 11
  v2 eq "Ontology"

  // 'neq' means "not equal"
  v2 neq "Epistemology"

  // [Error] Epistemology != Ontology
  // v2 eq "Epistemology"
}
/* Output:
11
Ontology
Ontology
*/
```

`atomictest` paketi için kod Ek A: AtomicTest'de bulunmaktadır. `AtomicTest.kt` dosyasındaki her şeyi şu anda anlamanızı beklemiyoruz, çünkü kitabın ilerleyen bölümlerine kadar ortaya çıkmayacak bazı özellikler kullanılıyor.

Temiz ve rahat bir görünüm sağlamak için `AtomicTest`, henüz görmediğiniz bir Kotlin özelliğini kullanır: `a.function(b)` şeklindeki bir fonksiyon çağrısını, metin benzeri formda `a function b` yazma yeteneği. Buna *infix gösterimi* denir. Sadece `infix` anahtar kelimesiyle tanımlanan fonksiyonlar bu şekilde çağrılabilir. `AtomicTest.kt`, `TestingExample.kt` dosyasında kullanılan `infix` `eq` ve `neq` fonksiyonlarını tanımlar:

```
expression eq expected
expression neq expected
```

`eq` ve `neq` esnektir—test ifadesi olarak hemen hemen her şey çalışır. Eğer *expected* bir `String` ise, *expression* bir `Stringe` dönüştürülür ve iki `String` karşılaştırılır. Aksi takdirde, *expression* ve *expected* direkt olarak (önce dönüştürülmeden) karşılaştırılır. Her iki durumda da, *expression*'ın sonucu konsolda görünür, böylece program

çalıştığında bir şeyler görürsünüz. Testler başarılı olsa bile, yine de `eq` veya `neq`'in solunda sonucu görürsünüz. Eğer *expression* ve *expected* eşdeğer değilse, program çalıştığında `AtomicTest` bir hata gösterir.

`TestingExample.kt`'deki son test, hata çıktısını görmeniz için kasten başarısız olur. Eğer iki değer eşit değilse, Kotlin `[Error]` ile başlayan ilgili mesajı görüntüler. Son satırı yorumdan çıkarır ve yukarıdaki örneği çalıştırırsanız, tüm başarılı testlerden sonra şunları göreceksiniz:

```
[Error] Epistemology != Ontology
```

`v2` içinde saklanan gerçek değer, "beklenen" ifadede iddia edildiği gibi değildir. `AtomicTest` hem beklenen hem de gerçek değerlerin `String` temsillerini gösterir.

`eq` ve `neq`, `AtomicTest` için tanımlanmış temel (infix) fonksiyonlardır—gerçekten de minimal bir test sistemidir. Örneklerinizde `eq` ve `neq` ifadelerini kullandığınızda, hem bir test hem de bazı konsol çıktıları oluşturacaksınız. Programın doğruluğunu çalıştırarak doğrularsınız.

`AtomicTest` içinde ikinci bir araç daha vardır. `trace` nesnesi, karşılaştırma için çıktıyı yakalar:

```
// Testing/Trace1.kt
import atomictest.*

fun main() {
  trace("line 1")
  trace(47)
  trace("line 2")
  trace eq """
    line 1
    47
    line 2
  """
}
```

Sonuçların `trace`e eklenmesi bir fonksiyon çağrısı gibi görünür, bu nedenle `println()` fonksiyonunu etkili bir şekilde `trace()` ile değiştirebilirsiniz.

Önceki atomlarda çıktıyı görüntüledik ve herhangi bir tutarsızlığı yakalamak için insan görsel incelemesine güvendik. Bu güvenilmez; kodu defalarca gözden geçirdiğimiz bir kitapta bile, görsel incelemenin hataları bulmak için güvenilir olmadığını

öğrendik. Artık `AtomicTest` her şeyi bizim için yapacağı için nadiren yorumlanmış çıktı blokları kullanıyoruz. Ancak, bazen daha yararlı bir etki sağladığında yorumlanmış çıktı bloklarını yine de dahil ediyoruz.

Kitabın geri kalanında test kullanmanın faydalarını görmek, testleri programlama sürecinize dahil etmenize yardımcı olacaktır. Muhtemelen testleri olmayan kodu gördüğünüzde rahatsızlık hissetmeye başlayacaksınız. Hatta testleri olmayan kodun tanım gereği bozuk olduğuna karar verebilirsiniz.

Programlamanın Bir Parçası Olarak Test Etme

Test etme, yazılım geliştirme sürecinize dahil edildiğinde en etkili olur. Test yazmak, beklediğiniz sonuçları almanızı sağlar. Birçok kişi, uygulama kodunu yazmadan *önce* testleri yazmayı savunur—önce testin başarısız olmasını sağlarsınız, ardından testi geçirecek kodu yazarsınız. Bu teknik, *Test Güdümlü Geliştirme* (TGG) olarak adlandırılır ve gerçekten neyi test ettiğinizi garanti etmenin bir yoludur. TGG'nin daha ayrıntılı bir açıklamasını Wikipedia'da bulabilirsiniz ("Test Driven Development" olarak arayın).

Test edilebilir şekilde yazmanın bir başka faydası da kodunuzu oluşturma şeklinizi değiştirmesidir. Sonuçları yalnızca konsolda görüntüleyebilirsiniz. Ancak test zihniyetinde, "Bunu nasıl test edeceğim?" diye düşünürsünüz. Bir fonksiyon oluşturduğunuzda, yalnızca bu sonucu test etmek için bile olsa fonksiyondan bir şey döndürmeniz gerektiğine karar verirsiniz. Girdi alıp çıktı üreten fonksiyonlar genellikle daha iyi tasarımlar oluşturur.

İşte Sayı Türleri'nden BMI hesaplamasını uygulamak için TGG kullanarak basitleştirilmiş bir örnek. Önce testleri yazarız, bu arada başarısız olan ilk uygulamayı (çünkü fonksiyonu henüz uygulamadık) yazarız:

```
// Testing/TDDFail.kt
package testing1
import atomictest.eq

fun main() {
  calculateBMI(160, 68) eq "Normal weight"
//  calculateBMI(100, 68) eq "Underweight"
//  calculateBMI(200, 68) eq "Overweight"
}

fun calculateBMI(lbs: Int, height: Int) =
  "Normal weight"
```

Sadece ilk test geçiyor. Diğer testler başarısız ve yorumlanmış durumda. Şimdi, hangi ağırlıkların hangi kategorilerde olduğunu belirlemek için kod ekliyoruz. Şimdi *tüm* testler başarısız oluyor:

```
// Testing/TDDStillFails.kt
package testing2
import atomictest.eq

fun main() {
  // Everything fails:
  // calculateBMI(160, 68) eq "Normal weight"
  // calculateBMI(100, 68) eq "Underweight"
  // calculateBMI(200, 68) eq "Overweight"
}

fun calculateBMI(
  lbs: Int,
  height: Int
): String {
  val bmi = lbs / (height * height) * 703.07
  return if (bmi < 18.5) "Underweight"
  else if (bmi < 25) "Normal weight"
  else "Overweight"
}
```

`Int` yerine `Double` kullanıyoruz ve bu da sıfır sonucu üretiyor. Testler bizi doğru çözüme yönlendiriyor:

```
// Testing/TDDWorks.kt
package testing3
import atomictest.eq

fun main() {
  calculateBMI(160.0, 68.0) eq "Normal weight"
  calculateBMI(100.0, 68.0) eq "Underweight"
  calculateBMI(200.0, 68.0) eq "Overweight"
}

fun calculateBMI(
  lbs: Double,
  height: Double
): String {
  val bmi = lbs / (height * height) * 703.07
  return if (bmi < 18.5) "Underweight"
  else if (bmi < 25) "Normal weight"
  else "Overweight"
}
```

Ek testler eklemeyi seçebilirsiniz. Bu testler sınır koşulları için olabilir.

Bu kitabın egzersizlerinde, kodunuzun geçmesi gereken testler bulunmaktadır.

Egzersizler ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

İstisnalar

> "İstisna" kelimesi, "Buna itiraz ediyorum." ifadesindeki anlamıyla kullanılır.

Olağanüstü bir durum, mevcut fonksiyonun veya scope'un devam etmesini engeller. Sorun meydana geldiğinde, ne yapacağınızı bilemeyebilirsiniz, ancak mevcut bağlamda devam edemezsiniz. Sorunu çözmek için yeterli bilgiye sahip değilsinizdir. Bu yüzden durmalı ve sorunu uygun eylemi gerçekleştirebilecek başka bir bağlama devretmelisiniz.

Bu bölüm, *istisnaları* bir hata raporlama mekanizması olarak ele alır. Bölüm VI: Başarısızlığı Önleme'de, sorunlarla başa çıkmanın diğer yollarını inceliyoruz.

Olağanüstü bir durumu normal bir sorundan ayırt etmek önemlidir. Normal bir sorun, mevcut bağlamda sorunla başa çıkmak için yeterli bilgiye sahiptir. Olağanüstü bir durumda ise işlemeye devam edemezsiniz. Yapabileceğiniz tek şey, sorunu dış bir bağlama devretmektir. Bu, *istisna fırlattığınızda* gerçekleşir. İstisna, hatanın meydana geldiği yerden "fırlatılan" nesnedir.

`toInt()`, bir `String`i `Int`e dönüştüren fonksiyonu düşünün. Bu fonksiyonu, bir tam sayı değeri içermeyen bir `String` için çağırırsanız ne olur?

```
// Exceptions/ToIntException.kt
package exceptions

fun erroneousCode() {
  // Uncomment this line to get an exception:
  // val i = "1$".toInt()        // [1]
}

fun main() {
  erroneousCode()
}
```

[1] nolu satırın yorumdan çıkarılması bir istisnaya neden olur. Burada, başarısız olan satır yorumlanmıştır, böylece kitabın inşasını durdurmayız, bu da her örneğin beklendiği gibi derlenip çalıştırılıp çalıştırılmadığını kontrol eder.

Bir istisna atıldığında, yürütme yolu—devam edilemeyen yol—durur ve istisna nesnesi mevcut bağlamdan çıkar. Burada, `erroneousCode()` bağlamından çıkar ve `main()` bağlamına gider. Bu durumda, Kotlin sadece hatayı rapor eder; programcı muhtemelen bir hata yapmıştır ve kodu düzeltmelidir.

Bir istisna yakalanmadığında, program durur ve ayrıntılı bilgiler içeren bir *yığın izi* görüntüler. `ToIntException.kt` dosyasındaki **[1]** numaralı satırı yorumdan çıkarmak, aşağıdaki çıktıyı üretir:

```
Exception in thread "main" java.lang.NumberFormatException: For input s\
tring: "1$"
  at java.lang.NumberFormatException.forInputString(NumberFormatExcepti\
on.java:65)
  at java.lang.Integer.parseInt(Integer.java:580)
  at java.lang.Integer.parseInt(Integer.java:615)
  at ToIntExceptionKt.erroneousCode(at ToIntException.kt:6)
  at ToIntExceptionKt.main(at ToIntException.kt:10)
```

Yığın izleme, hatanın oluştuğu dosya ve satır gibi ayrıntılar verir, böylece sorunu hızla keşfedebilirsiniz. Son iki satır sorunu gösterir: `main()` fonksiyonunun 10. satırında `erroneousCode()` çağrılır. Daha sonra, daha kesin olarak, `erroneousCode()` fonksiyonunun 6. satırında `toInt()` çağrılır.

İstisnaları görüntülemek için kodu yorumlamayı ve yorumdan çıkarmayı önlemek adına, `AtomicTest` paketinden `capture()` fonksiyonunu kullanırız:

```
// Exceptions/IntroducingCapture.kt
import atomictest.*

fun main() {
  capture {
    "1$".toInt()
  } eq "NumberFormatException: " +
    """For input string: "1$""""
}
```

`capture()` kullanarak, oluşturulan istisnayı beklenen hata mesajıyla karşılaştırıyoruz. `capture()` normal programlama için pek kullanışlı değildir—bu kitap için

özel olarak tasarlanmıştır, böylece istisnayı görebilir ve çıktının kitabın yapı sistemi tarafından kontrol edildiğini bilebilirsiniz.

Beklenen sonucu başarılı bir şekilde üretemediğinizde başvurabileceğiniz başka bir strateji de "değer yok" anlamına gelen özel bir sabit olan `null` döndürmektir. Herhangi bir türdeki bir değerin yerine `null` döndürebilirsiniz. Daha sonra Nullable Types bölümünde, `null`'ın ortaya çıkan ifadenin türünü nasıl etkilediğini tartışıyoruz.

Kotlin standart kütüphanesi, `String` bir tamsayı içeriyorsa dönüşümü gerçekleştiren veya dönüşüm imkansızsa `null` üreten `String.toIntOrNull()` içerir—`null` başarısızlığı belirtmenin basit bir yoludur:

```
// Exceptions/IntroducingNull.kt
import atomictest.eq

fun main() {
  "1$".toIntOrNull() eq null
}
```

Diyelim ki birkaç aylık bir dönem boyunca ortalama gelir hesaplıyoruz:

```
// Exceptions/AverageIncome.kt
package firstversion
import atomictest.*

fun averageIncome(income: Int, months: Int) =
  income / months

fun main() {
  averageIncome(3300, 3) eq 1100
  capture {
    averageIncome(5000, 0)
  } eq "ArithmeticException: / by zero"
}
```

Eğer `months` sıfır ise, `averageIncome()`'deki bölme işlemi bir `ArithmeticException` fırlatır. Ne yazık ki, bu bize hatanın neden oluştuğu, paydanın ne anlama geldiği ve ilk etapta sıfır olup olamayacağı hakkında hiçbir şey söylemez. Bu, açıkça kodda bir hatadır—`averageIncome()` bir `0` `months` değeriyle başa çıkmalı ve sıfıra bölme hatasını önlemelidir.

Sorunun kaynağı hakkında daha fazla bilgi üretmesi için `averageIncome()`'i değiştirelim. Eğer `months` sıfır ise, sonuç olarak normal bir tamsayı değeri döndüremeyiz. Bir strateji, `null` döndürmektir:

```
// Exceptions/AverageIncomeWithNull.kt
package withnull
import atomictest.eq

fun averageIncome(income: Int, months: Int) =
  if (months == 0)
    null
  else
    income / months

fun main() {
  averageIncome(3300, 3) eq 1100
  averageIncome(5000, 0) eq null
}
```

Eğer bir fonksiyon `null` döndürebiliyorsa, Kotlin bu sonucu kullanmadan önce kontrol etmenizi gerektirir (bu konu Nullable Türler kısmında ele alınmıştır). Kullanıcıya sadece çıktı göstermek isteseniz bile, "Tam ay dönemleri geçmedi," demek, "Dönem için ortalama geliriniz: null." demekten daha iyidir.

`averageIncome()` fonksiyonunu yanlış argümanlarla yürütmek yerine, bir istisna fırlatabilirsiniz—bu sorunu yönetmek için programın başka bir kısmına kaçış ve zorlama yapmış olursunuz. Varsayılan `ArithmeticException`'ı kabul edebilirsiniz, ancak genellikle ayrıntılı bir hata mesajıyla belirli bir istisna fırlatmak daha yararlıdır. Uygulamanız, üretimde birkaç yıl sonra yeni bir özellik `averageIncome()` fonksiyonunu düzgün argümanlar kontrol edilmeden çağırdığı için aniden bir istisna fırlattığında, o mesaj için minnettar olacaksınız.

```
// Exceptions/AverageIncomeWithException.kt
package properexception
import atomictest.*

fun averageIncome(income: Int, months: Int) =
  if (months == 0)
    throw IllegalArgumentException(    // [1]
      "Months can't be zero")
  else
    income / months

fun main() {
  averageIncome(3300, 3) eq 1100
  capture {
    averageIncome(5000, 0)
  } eq "IllegalArgumentException: " +
    "Months can't be zero"
}
```

- **[1]** Bir istisna fırlatırken, `throw` anahtar kelimesi fırlatılacak istisnanın ardından gelir ve gerekirse herhangi bir argümanla birlikte kullanılır. Burada standart istisna sınıfı `IllegalArgumentException` kullanıyoruz.

Amacınız, gelecekte uygulamanızın desteğini basitleştirmek için mümkün olan en kullanışlı mesajları oluşturmaktır. Daha sonra kendi istisna türlerinizi tanımlamayı ve bunları kendi durumunuza özgü hale getirmeyi öğreneceksiniz.

Egzersizler ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Listeler

> Bir `List`, diğer nesneleri tutan bir nesne olan bir *kapsayıcı*dır.

Kapsayıcılar aynı zamanda *koleksiyonlar* olarak da adlandırılır. Bu kitaptaki örneklerde temel bir kapsayıcıya ihtiyacımız olduğunda, genellikle bir `List` kullanırız.

`List`ler standart Kotlin paketinin bir parçasıdır, bu yüzden bir `import` gerektirmezler.

Aşağıdaki örnek, başlangıç değerleriyle `listOf()` standart kütüphane fonksiyonunu çağırarak `Int`lerle doldurulmuş bir `List` oluşturur:

```
// Lists/Lists.kt
import atomictest.eq

fun main() {
  val ints = listOf(99, 3, 5, 7, 11, 13)
  ints eq "[99, 3, 5, 7, 11, 13]"   // [1]

  // Select each element in the List:
  var result = ""
  for (i in ints) {                 // [2]
    result += "$i "
  }
  result eq "99 3 5 7 11 13"

  // "Indexing" into the List:
  ints[4] eq 11                     // [3]
}
```

- **[1]** Bir `List` kendini görüntülerken köşeli parantezler kullanır.
- **[2]** `for` döngüleri `List`ler ile iyi çalışır: `for(i in ints)` ifadesi `i`'nin `ints` içerisindeki her değeri alması anlamına gelir. `val i`'yi deklare etmezsiniz veya türünü belirtmezsiniz; Kotlin bağlamdan `i`'nin bir `for` döngüsü tanımlayıcısı olduğunu bilir.

- **[3]** Köşeli parantezler bir `List` içinde *indeksler*. Bir `List` öğelerini başlatma sırasına göre tutar ve onları tek tek numaralarla seçersiniz. Çoğu programlama dilinde olduğu gibi, Kotlin sıfırıncı öğeden indekslemeye başlar, bu durumda `99` değerini üretir. Dolayısıyla `4` indeksi `11` değerini üretir.

İndekslemenin sıfırdan başladığını unutmak, sözde *bir-eksik* hatasına yol açar. Kotlin gibi bir dilde genellikle öğeleri tek tek seçmeyiz, bunun yerine `in` kullanarak tüm bir konteyner boyunca *yineleme* yaparız. Bu, bir-eksik hatalarını ortadan kaldırır.

Bir `List` içindeki son öğenin ötesinde bir indeks kullanırsanız, Kotlin `ArrayIndexOutOfBoundsException` hatası fırlatır:

```
// Lists/OutOfBounds.kt
import atomictest.*

fun main() {
  val ints = listOf(1, 2, 3)
  capture {
    ints[3]
  } contains
    listOf("ArrayIndexOutOfBoundsException")
}
```

Bir `List` farklı türleri barındırabilir. İşte bir `Double` `List`i ve bir `String` `List`i:

```
// Lists/ListUsefulFunction.kt
import atomictest.eq

fun main() {
  val doubles =
    listOf(1.1, 2.2, 3.3, 4.4)
  doubles.sum() eq 11.0

  val strings = listOf("Twas", "Brillig",
    "And", "Slithy", "Toves")
  strings eq listOf("Twas", "Brillig",
    "And", "Slithy", "Toves")
  strings.sorted() eq listOf("And",
    "Brillig", "Slithy", "Toves", "Twas")
  strings.reversed() eq listOf("Toves",
    "Slithy", "And", "Brillig", "Twas")
  strings.first() eq "Twas"
```

```
  strings.takeLast(2) eq
    listOf("Slithy", "Toves")
}
```

Bu, `List`'in bazı operasyonlarını gösterir. "sort" yerine "sorted" adını not edin. `sorted()` çağırdığınızda, aynı öğeleri içeren yeni bir `List` *oluşturur*, ancak orijinal `List`'i olduğu gibi bırakır. Buna "sort" demek, orijinal `List`'in doğrudan değiştirildiğini ima eder (yani, *yerinde sıralanmış*). Kotlin boyunca, "orijinal nesneyi olduğu gibi bırakma ve yeni bir nesne oluşturma" eğilimini görürsünüz. `reversed()` da yeni bir `List` oluşturur.

Parametreli Türler

Tip çıkarımı kullanmanın iyi bir uygulama olduğunu düşünüyoruz — bu, kodu daha temiz ve okunması daha kolay hale getirme eğilimindedir. Ancak bazen Kotlin, hangi türü kullanacağını anlayamadığı konusunda şikayet eder ve diğer durumlarda açıklık, kodu daha anlaşılır kılar. İşte Kotlin'e bir `List` tarafından içerilen türü nasıl söylediğimiz:

```
// Lists/ParameterizedTypes.kt
import atomictest.eq

fun main() {
  // Type is inferred:
  val numbers = listOf(1, 2, 3)
  val strings =
    listOf("one", "two", "three")
  // Exactly the same, but explicitly typed:
  val numbers2: List<Int> = listOf(1, 2, 3)
  val strings2: List<String> =
    listOf("one", "two", "three")
  numbers eq numbers2
  strings eq strings2
}
```

Kotlin, `numbers`'ın bir `Int` içeren bir `Liste` ve `strings`'in bir `String` içeren bir `Liste` olduğunu çıkarmak için başlatma değerlerini kullanır.

`numbers2` ve `strings2`, `numbers` ve `strings`'in açıkça tip belirlenmiş versiyonlarıdır ve tip deklarasyonları `List<Int>` ve `List<String>` eklenerek oluşturulmuştur. Daha önce açılı parantezler görmemiştiniz—bunlar bir *tip parametresi* belirtir, bu sayede "bu kap 'parametre' nesnelerini tutar" diyebilirsiniz. `List<Int>`'i "`Int` içeren `Liste`" olarak telaffuz ederiz.

Tip parametreleri kaplardan başka bileşenler için de kullanışlıdır, ancak genellikle kap benzeri nesnelerde görürsünüz.

Dönüş değerleri de tip parametrelerine sahip olabilir:

```
// Lists/ParameterizedReturn.kt
package lists
import atomictest.eq

// Return type is inferred:
fun inferred(p: Char, q: Char) =
  listOf(p, q)

// Explicit return type:
fun explicit(p: Char, q: Char): List<Char> =
  listOf(p, q)

fun main() {
  inferred('a', 'b') eq "[a, b]"
  explicit('y', 'z') eq "[y, z]"
}
```

Kotlin, `inferred()` için dönüş tipini tahmin ederken, `explicit()` fonksiyon dönüş tipini belirtir. Sadece bir `List` döndüğünü söyleyemezsiniz; Kotlin itiraz eder, bu yüzden tip parametresini de vermelisiniz. Bir fonksiyonun dönüş tipini belirttiğinizde, Kotlin niyetinizi zorunlu kılar.

Salt Okunur ve Değiştirilebilir Listeler

Eğer açıkça değiştirilebilir bir `List` istediğinizi belirtmezseniz, bir tane alamazsınız. `listOf()`, mutasyona uğramayan fonksiyonları olan salt okunur bir `Liste` üretir.

Bir `Listi` kademeli olarak oluşturuyorsanız (yani, oluşturma zamanında tüm elemanlara sahip değilseniz), `mutableListOf()` kullanın. Bu, değiştirilebilen bir `DeğiştirilebilirListe` üretir:

```
// Lists/MutableList.kt
import atomictest.eq

fun main() {
  val list = mutableListOf<Int>()

  list.add(1)
  list.addAll(listOf(2, 3))

  list += 4
  list += listOf(5, 6)

  list eq listOf(1, 2, 3, 4, 5, 6)
}
```

`list`'in başlangıçta hiç elemanı olmadığı için, Kotlin'e hangi tür olduğunu söylememiz gerekiyor. Bunu `mutableListOf()` çağrısında `<Int>` belirtimi yaparak sağlıyoruz. Bir `MutableList`'e `add()` ve `addAll()` kullanarak veya tek bir eleman ya da başka bir koleksiyon ekleyen `+=` operatörünü kullanarak eleman ekleyebilirsiniz.

Bir `MutableList`, bir `List` olarak ele alınabilir, bu durumda değiştirilemez. Ancak, salt okunur bir `List`'i bir `MutableList` olarak ele alamazsınız:

```
// Lists/MutListIsList.kt
package lists
import atomictest.eq

fun makeList(): List<Int> =
  mutableListOf(1, 2, 3)

fun main() {
  // makeList() produces a read-only List:
  val list = makeList()
  // list.add(3) // Unresolved reference: add
  list eq listOf(1, 2, 3)
}
```

`list`, `makeList()` içinde `mutableListOf()` kullanılarak oluşturulmasına rağmen mutasyon fonksiyonlarına sahip değildir. `makeList()` fonksiyonunun dönüş tipi `List<Int>` olarak belirtilmiştir. Orijinal nesne hala bir `MutableList` olsa da, `List` perspektifinden görülmektedir.

Bir `List` *salt okunur*dur—içeriğini okuyabilirsiniz ama yazamazsınız. Eğer altta yatan uygulama bir `MutableList` ise ve bu uygulamaya yönelik değiştirilebilir bir referansı korursanız, bu değiştirilebilir referans aracılığıyla hala değişiklik yapabilirsiniz ve herhangi bir salt okunur referans bu değişiklikleri görecektir. Bu, Görünürlüğü Sınırlandırma bölümünde tanıtılan *ad takma* olgusunun bir başka örneğidir:

```
// Lists/MultipleListRefs.kt
import atomictest.eq

fun main() {
  val first = mutableListOf(1)
  val second: List<Int> = first
  second eq listOf(1)
  first.add(2)
  // second sees the change:
  second eq listOf(1, 2)
}
```

`first`, `mutableListOf(1)` tarafından üretilen değiştirilebilir nesneye yapılan değişmez (`val`) bir referanstır. `second`'a `first`'ün takma adı verildiğinde, aynı nesnenin bir görünümü haline gelir. `second`, değiştirme işlevlerini içermediği için salt okunurdur çünkü `List<Int>` değişiklik işlevlerini içermez. Açık `List<Int>` türü bildirimi olmadan, Kotlin `second`'un da değiştirilebilir bir nesneye referans olduğunu çıkarırdı.

`first` değiştirilebilir bir `Liste` referans olduğu için nesneye bir eleman (2) ekleyebiliriz. `second`'un bu değişiklikleri gözlemlediğini unutmayın—`List`'i değiştiremez ancak `List`, `first` aracılığıyla değişir.

`+ =` Bulmacası

`+ =` operatörü, değişmez bir `List`'in aslında değiştirilebilir olduğu izlenimini verebilir:

```
// Lists/ApparentlyMutableList.kt
import atomictest.eq

fun main() {
  var list = listOf('X') // Immutable
  list += 'Y' // Appears to be mutable
  list eq "[X, Y]"
}
```

`listOf()` değişmez (immutable) bir `List` üretir, ancak `list + = 'Y'` bu `List`'i değiştiriyor gibi görünüyor. `+ =` işareti bir şekilde değişmezliği ihlal ediyor mu?

Bu sadece `list` bir `var` olduğunda gerçekleşir. İşte `val`/`var` ile değişebilir/değişmez `List`'lerin farklı kombinasyonlarını gösteren daha ayrıntılı bir örnek:

```
// Lists/PlusAssignPuzzle.kt
import atomictest.eq

fun main() {
    // Mutable List assigned to a 'val'/'var':
    val list1 = mutableListOf('A') // or 'var'
    list1 += 'A' // Is the same as:
    list1.plusAssign('A')              // [1]

    // Immutable List assigned to a 'val':
    val list2 = listOf('B')
    // list2 + = 'B' // Is the same as:
    // list2 = list2 + 'B'             // [2]

    // Immutable List assigned to a 'var':
    var list3 = listOf('C')
    list3 += 'C' // Is the same as:
    val newList = list3 + 'C'          // [3]
    list3 = newList                    // [4]

    list1 eq "[A, A, A]"
    list2 eq "[B]"
    list3 eq "[C, C, C]"
}
```

- **[1]** `list1` değiştirilebilir bir nesneyi ifade eder, bu nedenle yerinde değiştirilebilir. Derleyici `+ =` ifadesini `plusAssign()` çağrısına çevirir. `list1`'in bir `val`

ya da `var` olup olmadığı önemli değildir çünkü hiçbir zaman *yeniden atanmaz*; oluşturulduktan sonra her zaman aynı değiştirilebilir listeyi ifade eder. Onu bir `var` yapın ve IntelliJ değişmediğini belirtir ve bir `val` olması gerektiğini önerir.

- **[2]** Bu, `list2` ve `'B'`'yi birleştirerek yeni bir `List` oluşturmaya çalışır, ancak `list2` bir `val` olduğu için bu yeni `List`'i `list2`'ye yeniden atayamaz. Bu yeniden atama yeteneği olmadan, `+ =` derlenemez.
- **[3]** `newListi` mevcut değiştirilemez `Listi`, `list3` tarafından ifade edilen, değiştirmeden oluşturur.
- **[4]** `list3` bir `var` olduğu için, derleyici `newListi` `list3`'e geri atar. `list3`'ün önceki içeriği unutulur ve `list3`'ün değiştirildiği görünür. Aslında, eski `list3` atılmış ve yeni oluşturulan `newList` ile değiştirilmiştir, bu da `list3`'ün değiştirilebilir olduğu yanılsamasını verir.

Bu `+ =` davranışı diğer koleksiyonlarla da meydana gelir. Ortaya çıkan karmaşa, tanımlayıcılarınız için `val` yerine `var` seçmenizin bir başka nedenidir.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Değişken Argüman Listeleri

> `vararg` anahtar kelimesi esnek boyutlu bir argüman listesi üretir.

Listeler bölümünde, herhangi bir sayıda parametre alıp bir `List` üreten `listOf()` fonksiyonunu tanıtmıştık:

```
// Varargs/ListOf.kt
import atomictest.eq

fun main() {
  listOf(1) eq "[1]"
  listOf("a", "b") eq "[a, b]"
}
```

`vararg` anahtar kelimesini kullanarak, tıpkı `listOf()` fonksiyonunda olduğu gibi, herhangi bir sayıda argüman alan bir fonksiyon tanımlayabilirsiniz. `vararg`, *değişken argüman listesi*nin kısaltmasıdır:

```
// Varargs/VariableArgList.kt
package varargs

fun v(s: String, vararg d: Double) {}

fun main() {
  v("abc", 1.0, 2.0)
  v("def", 1.0, 2.0, 3.0, 4.0)
  v("ghi", 1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
}
```

Bir fonksiyon tanımı yalnızca bir parametreyi `vararg` olarak belirtebilir. Parametre listesindeki herhangi bir öğeyi `vararg` olarak belirtmek mümkün olsa da, genellikle sonuncusunu yapmak en basitidir.

`vararg`, herhangi bir sayıda (sıfır dahil) argümanı geçirmenize olanak tanır. Tüm argümanlar belirtilen türde olmalıdır. `vararg` argümanlarına, bir `Array` haline gelen parametre adı kullanılarak erişilir:

```
// Varargs/VarargSum.kt
package varargs
import atomictest.eq

fun sum(vararg numbers: Int): Int {
  var total = 0
  for (n in numbers) {
    total += n
  }
  return total
}

fun main() {
  sum(13, 27, 44) eq 84
  sum(1, 3, 5, 7, 9, 11) eq 36
  sum() eq 0
}
```

`Array` ve `List` her ne kadar benzer görünseler de, farklı şekillerde uygulanmışlardır—`List` normal bir kütüphane sınıfıyken, `Array` özel düşük seviyeli desteğe sahiptir. `Array`, Kotlin'in diğer dillerle, özellikle de Java ile uyumluluk gereksiniminden gelir.

Günlük programlamada, basit bir sıralamaya ihtiyacınız olduğunda bir `List` kullanın. Sadece üçüncü parti bir API'nin `Array` gerektirdiği durumlarda veya `vararg` ile uğraşıyorsanız `Array` kullanın.

Çoğu durumda `vararg`'ın bir `Array` ürettiği gerçeğini göz ardı edebilir ve onu bir `List` gibi davranabilirsiniz:

```
// Varargs/VarargLikeList.kt
package varargs
import atomictest.eq

fun evaluate(vararg ints: Int) =
  "Size: ${ints.size}\n" +
  "Sum: ${ints.sum()}\n" +
  "Average: ${ints.average()}"

fun main() {
  evaluate(10, -3, 8, 1, 9) eq """
    Size: 5
    Sum: 25
```

```
    Average: 5.0
  """
}
```

vararg kabul edilen her yere bir Dizi öğesi geçirebilirsiniz. Bir Dizi oluşturmak için, listOf() kullandığınız şekilde arrayOf() kullanın. Bir Dizi her zaman değiştirilebilir. Bir Dizi'yi bir dizi argümana (sadece Dizi türünde tek bir öğe değil) dönüştürmek için *yayılma operatörünü*, * kullanın:

```
// Varargs/SpreadOperator.kt
import varargs.sum
import atomictest.eq

fun main() {
  val array = intArrayOf(4, 5)
  sum(1, 2, 3, *array, 6) eq 21  // [1]
  // Doesn't compile:
  // sum(1, 2, 3, array, 6)

  val list = listOf(9, 10, 11)
  sum(*list.toIntArray()) eq 30  // [2]
}
```

Eğer yukarıdaki örnekte olduğu gibi ilkel türler (Int, Double veya Boolean gibi) içeren bir Array geçirirseniz, Array oluşturma fonksiyonu özellikle tip belirtilmelidir. arrayOf(4, 5) yerine intArrayOf(4, 5) kullanırsanız, [1] numaralı satır *çıkarılan tür Array<Int> ama IntArray bekleniyordu* hatası verecektir.

Yayılma operatörü sadece dizilerle çalışır. Eğer bir List'i argüman dizisi olarak geçirmek istiyorsanız, önce onu bir Array'e dönüştürün ve ardından yayılma operatörünü uygulayın, **[2]**'deki gibi. Sonuç bir ilkel tür Array olduğu için, yine toIntArray() dönüşüm fonksiyonunu kullanmalıyız.

Yayılma operatörü, başka bir fonksiyona vararg argümanları geçirmeniz gerektiğinde özellikle yararlıdır:

```
// Varargs/TwoFunctionsWithVarargs.kt
package varargs
import atomictest.eq

fun first(vararg numbers: Int): String {
  var result = ""
  for (i in numbers) {
    result += "[$i]"
  }
  return result
}

fun second(vararg numbers: Int) =
  first(*numbers)

fun main() {
  second(7, 9, 32) eq "[7][9][32]"
}
```

Komut Satırı Argümanları

Bir programı komut satırında çalıştırırken, ona değişken sayıda argüman geçirebilirsiniz. Komut satırı argümanlarını yakalamak için, `main()` fonksiyonuna belirli bir parametre sağlamalısınız:

```
// Varargs/MainArgs.kt

fun main(args: Array<String>) {
  for (a in args) {
    println(a)
  }
}
```

Parametre geleneksel olarak `args` olarak adlandırılır (ancak istediğiniz herhangi bir şeyi çağırabilirsiniz) ve `args` için tür yalnızca `Array<String>` (`String` türünde `Array`) olabilir.

IntelliJ IDEA kullanıyorsanız, program argümanlarını bu atom için son alıştırmada gösterildiği gibi ilgili "Çalıştırma yapılandırması"nı düzenleyerek geçirebilirsiniz.

Ayrıca `kotlinc` derleyicisini kullanarak bir komut satırı programı üretebilirsiniz. Bilgisayarınızda `kotlinc` yoksa, Kotlin ana sitesindeki[27] talimatları izleyin. `MainArgs.kt` kodunu girdikten ve kaydettikten sonra, bir komut isteminde aşağıdakileri yazın:

```
kotlinc MainArgs.kt
```

Komut satırı argümanlarını program çağrısını takip edecek şekilde şu şekilde sağlarsınız:

```
kotlin MainArgsKt hamster 42 3.14159
```

Bu çıktıyı göreceksiniz:

```
hamster
42
3.14159
```

Bir `String` parametresini belirli bir türe dönüştürmek istiyorsanız, Kotlin gibi dönüşüm fonksiyonları sağlar. Örneğin, bir `Int`'e dönüştürmek için `toInt()`, ve bir `Float`'a dönüştürmek için `toFloat()`. Bunları kullanmak, komut satırı argümanlarının belirli bir sırada görünmesini varsayar. Burada, program bir `String`, ardından bir `Int`'e dönüştürülebilecek bir şey, ve ardından bir `Float`'a dönüştürülebilecek bir şey bekler:

```
// Varargs/MainArgConversion.kt

fun main(args: Array<String>) {
  if (args.size < 3) return
  val first = args[0]
  val second = args[1].toInt()
  val third = args[2].toFloat()
  println("$first  $second  $third")
}
```

`main()` içindeki ilk satır yeterli argüman yoksa programdan çıkış yapar. Eğer ikinci ve üçüncü komut satırı argümanları olarak bir `Int` ve bir `Float`'a dönüştürülebilir bir şey sağlamazsanız, çalışma zamanı hataları görürsünüz (hataları görmek için deneyin).

[27] https://kotlinlang.org/

`MainArgConversion.kt` dosyasını daha önce kullandığımız aynı komut satırı argümanları ile derleyin ve çalıştırın, ve şunu göreceksiniz:

```
hamster 42 3.14159
```

Alıştırmalar ve çözümleri www.AtomicKotlin.com adresinde bulunabilir.

Kümeler

Bir `Set`, her değerden yalnızca bir öğeye izin veren bir koleksiyondur.

En yaygın `Set` aktivitesi, `in` veya `contains()` kullanarak üyeliği test etmektir:

```
// Sets/Sets.kt
import atomictest.eq

fun main() {
  val intSet = setOf(1, 1, 2, 3, 9, 9, 4)
  // No duplicates:
  intSet eq setOf(1, 2, 3, 4, 9)

  // Element order is unimportant:
  setOf(1, 2) eq setOf(2, 1)

  // Set membership:
  (9 in intSet) eq true
  (99 in intSet) eq false

  intSet.contains(9) eq true
  intSet.contains(99) eq false

  // Does this set contain another set?
  intSet.containsAll(setOf(1, 9, 2)) eq true

  // Set union:
  intSet.union(setOf(3, 4, 5, 6)) eq
    setOf(1, 2, 3, 4, 5, 6, 9)

  // Set intersection:
  intSet intersect setOf(0, 1, 2, 7, 8) eq
    setOf(1, 2)

  // Set difference:
  intSet subtract setOf(0, 1, 9, 10) eq
```

```
    setOf(2, 3, 4)
  intSet - setOf(0, 1, 9, 10) eq
    setOf(2, 3, 4)
}
```

Bu örnek şunları gösteriyor:

1. Yinelenen öğeleri bir `Set` içine yerleştirmek, bu tekrarları otomatik olarak kaldırır.
2. Kümelerde öğe sırası önemli değildir. Aynı öğeleri içeriyorlarsa iki küme eşittir.
3. Hem `in` hem de `contains()` üyelik testi yapar.
4. Alt küme kontrolü, birleşim, kesişim ve fark gibi Venn diyagramı işlemlerini ya nokta notasyonu (`set.union(other)`) ya da infiks notasyon (`set intersect other`) kullanarak gerçekleştirebilirsiniz. `union`, `intersect` ve `subtract` fonksiyonları infiks notasyon ile kullanılabilir.
5. Küme farkı, `subtract()` veya eksi operatörü ile ifade edilebilir.

Bir `List`'ten tekrarları kaldırmak için, onu bir `Set`'e dönüştürün:

```
// Sets/RemoveDuplicates.kt
import atomictest.eq

fun main() {
  val list = listOf(3, 3, 2, 1, 2)
  list.toSet() eq setOf(1, 2, 3)
  list.distinct() eq listOf(3, 2, 1)
  "abbcc".toSet() eq setOf('a', 'b', 'c')
}
```

Ayrıca, `distinct()` kullanabilirsiniz, bu da bir `List` döndürür. Bir `String` üzerinde `toSet()` çağırarak onu benzersiz karakterlerden oluşan bir sete dönüştürebilirsiniz.

`List` ile olduğu gibi, Kotlin `Set` için iki oluşturma fonksiyonu sağlar. `setOf()` fonksiyonunun sonucu salt okunurdur. Değiştirilebilir bir `Set` oluşturmak için, `mutableSetOf()` kullanın:

```
// Sets/MutableSet.kt
import atomictest.eq

fun main() {
  val mutableSet = mutableSetOf<Int>()
  mutableSet += 42
  mutableSet += 42
  mutableSet eq setOf(42)
  mutableSet -= 42
  mutableSet eq setOf<Int>()
}
```

Operatörler `+ =` ve `-=` `Set`lere eleman ekler ve çıkarır, tıpkı `List`lerde olduğu gibi.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Haritalar

> Bir `Map`, *anahtarları değerlere* bağlar ve bir anahtar verildiğinde bir değeri arar.

`mapOf()` fonksiyonuna anahtar-değer çiftleri sağlayarak bir `Map` oluşturursunuz. Her anahtarı ilgili değerinden `to` kullanarak ayırırız:

```
// Maps/Maps.kt
import atomictest.eq

fun main() {
  val constants = mapOf(
    "Pi" to 3.141,
    "e" to 2.718,
    "phi" to 1.618
  )
  constants eq
    "{Pi =3.141, e =2.718, phi =1.618}"

  // Look up a value from a key:
  constants["e"] eq 2.718              // [1]
  constants.keys eq setOf("Pi", "e", "phi")
  constants.values eq "[3.141, 2.718, 1.618]"

  var s = ""
  // Iterate through key-value pairs:
  for (entry in constants) {           // [2]
    s += "${entry.key}=${entry.value}, "
  }
  s eq "Pi =3.141, e =2.718, phi =1.618,"

  s = ""
  // Unpack during iteration:
  for ((key, value) in constants)      // [3]
    s += "$key=$value, "
  s eq "Pi =3.141, e =2.718, phi =1.618,"
}
```

- **[1]** `[]` operatörü, bir anahtar kullanarak bir değeri arar. `keys` kullanarak tüm anahtarları ve `values` kullanarak tüm değerleri üretebilirsiniz. `keys` çağrısı, bir `Map` içindeki tüm anahtarlar benzersiz olmalıdır, aksi takdirde bir arama sırasında belirsizlik olacağı için bir `Set` üretir.
- **[2]** Bir `Map` içinde dolaşmak, harita girişleri olarak anahtar-değer çiftleri üretir.
- **[3]** Dolaşırken anahtarları ve değerleri açabilirsiniz.

Düz bir `Map` salt okunurdur. İşte bir `MutableMap`:

```
// Maps/MutableMaps.kt
import atomictest.eq

fun main() {
  val m =
    mutableMapOf(5 to "five", 6 to "six")
  m[5] eq "five"
  m[5] = "5ive"
  m[5] eq "5ive"
  m += 4 to "four"
  m eq mapOf(5 to "5ive",
    4 to "four", 6 to "six")
}
```

`map[key] = value` `key` ile ilişkilendirilmiş `value`'yu ekler veya değiştirir. Ayrıca `map + = key to value` diyerek açıkça bir çift ekleyebilirsiniz.

`mapOf()` ve `mutableMapOf()` elemanların `Map` içine konma sırasını korur. Bu, diğer `Map` türleri için garanti edilmez.

Salt okunur bir `Map` değişikliklere izin vermez:

```
// Maps/ReadOnlyMaps.kt
import atomictest.eq

fun main() {
  val m = mapOf(5 to "five", 6 to "six")
  m[5] eq "five"
  // m[5] = "5ive" // Fails
  // m + = (4 to "four") // Fails
  m + (4 to "four") // Doesn't change m
  m eq mapOf(5 to "five", 6 to "six")
  val m2 = m + (4 to "four")
  m2 eq mapOf(
    5 to "five", 6 to "six", 4 to "four")
}
```

`m` tanımı, `Int`leri `String`lerle ilişkilendiren bir `Map` oluşturur. Bir `String`i değiştirmeye çalışırsak, Kotlin bir hata verir.

`+` ile yapılan bir ifade, hem eski öğeleri hem de yenisini içeren yeni bir `Map` oluşturur, ancak orijinal `Map`i etkilemez. Salt okunur bir `Map`e bir öğe "eklemenin" tek yolu, yeni bir `Map` oluşturmaktır.

Bir `Map`, belirli bir anahtar için bir giriş içermiyorsa `null` döner. `null` olamayacak bir sonuç gerekiyorsa, `getValue()` kullanın ve anahtar yoksa `NoSuchElementException` yakalayın:

```
// Maps/GetValue.kt
import atomictest.*

fun main() {
  val map = mapOf('a' to "attempt")
  map['b'] eq null
  capture {
    map.getValue('b')
  } eq "NoSuchElementException: " +
    "Key b is missing in the map."
  map.getOrDefault('a', "??") eq "attempt"
  map.getOrDefault('b', "??") eq "??"
}
```

`getOrDefault()` genellikle `null` veya bir istisnaya göre daha hoş bir alternatiftir.

Bir Map içinde sınıf örneklerini değer olarak saklayabilirsiniz. İşte bir numara String kullanarak bir Contact alan bir harita:

```
// Maps/ContactMap.kt
package maps
import atomictest.eq

class Contact(
  val name: String,
  val phone: String
) {
  override fun toString() =
    "Contact('$name', '$phone')"
}

fun main() {
  val miffy = Contact("Miffy", "1-234-567890")
  val cleo = Contact("Cleo", "098-765-4321")
  val contacts = mapOf(
    miffy.phone to miffy,
    cleo.phone to cleo)
  contacts["1-234-567890"] eq miffy
  contacts["1-111-111111"] eq null
}
```

Sınıf örneklerini anahtar olarak Map içinde kullanmak mümkündür, ancak bu biraz daha zordur, bu yüzden kitapta daha sonra tartışacağız.

- -

Mapler basit küçük veritabanları gibi görünür. Bazen *ilişkisel diziler* olarak adlandırılırlar, çünkü anahtarları değerlerle ilişkilendirirler. Tam özellikli bir veritabanına kıyasla oldukça sınırlı olsalar da, yine de şaşırtıcı derecede kullanışlıdırlar (ve bir veritabanından çok daha verimlidirler).

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Özellik Erişimcileri

> Bir özelliği okumak için, adını kullanın. Değiştirilebilir bir özelliğe değer atamak için, atama operatörü = kullanın.

Bu, `i` özelliğini okur ve yazar:

```
// PropertyAccessors/Data.kt
package propertyaccessors
import atomictest.eq

class Data(var i: Int)

fun main() {
  val data = Data(10)
  data.i eq 10 // Read the 'i' property
  data.i = 20  // Write to the 'i' property
}
```

Bu, `i` adlı depolama parçasına doğrudan erişim gibi görünüyor. Ancak, Kotlin okuma ve yazma işlemlerini gerçekleştirmek için fonksiyonlar çağırır. Beklediğiniz gibi, bu fonksiyonların varsayılan davranışı `i` içinde depolanan verileri okur ve yazar. Bu atomda, okuma ve yazma eylemlerini özelleştirmek için kendi *özellik erişimcilerinizi* yazmayı öğreneceksiniz.

Bir özelliğin değerini almak için kullanılan erişimciye *alıcı* denir. Bir alıcıyı, özellik tanımının hemen ardından `get()` tanımlayarak oluşturursunuz. Değiştirilebilir bir özelliği değiştirmek için kullanılan erişimciye *ayarlayıcı* denir. Bir ayarlayıcıyı, özellik tanımının hemen ardından `set()` tanımlayarak oluşturursunuz.

Aşağıdaki örnekte tanımlanan özellik erişimcileri, Kotlin tarafından oluşturulan varsayılan uygulamaları taklit eder. Özellik erişimcilerinin okuma ve yazma işlemleri sırasında gerçekten çağrıldığını görebilmeniz için ek bilgi gösteriyoruz. `get()` ve `set()` fonksiyonlarını görsel olarak özellik ile ilişkilendirmek için girintili yapıyoruz, ancak gerçek ilişki bu fonksiyonların özellikten hemen sonra tanımlanmış olmasından kaynaklanır (Kotlin girintiye önem vermez):

```
// PropertyAccessors/Default.kt
package propertyaccessors
import atomictest.*

class Default {
  var i: Int = 0
    get() {
      trace("get()")
      return field       // [1]
    }
    set(value) {
      trace("set($value)")
      field = value      // [2]
    }
}

fun main() {
  val d = Default()
  d.i = 2
  trace(d.i)
  trace eq """
    set(2)
    get()
    2
  """
}
```

`get()` ve `set()` tanımlama sırası önemli değildir. `get()`'i `set()` tanımlamadan tanımlayabilirsiniz ve tam tersi de geçerlidir.

Bir özelliğin varsayılan davranışı, saklanan değerini bir alıcıdan döndürür ve bir ayarlayıcı ile değiştirir—**[1]** ve **[2]** eylemleri. Alıcı ve ayarlayıcı içinde saklanan değer, yalnızca bu iki fonksiyon içinde erişilebilen `field` anahtar kelimesi kullanılarak dolaylı olarak manipüle edilir.

Bir sonraki örnek, alıcının varsayılan uygulamasını kullanır ve `n` özelliğindeki değişiklikleri izlemek için bir ayarlayıcı ekler:

```
// PropertyAccessors/LogChanges.kt
package propertyaccessors
import atomictest.*

class LogChanges {
  var n: Int = 0
    set(value) {
      trace("$field becomes $value")
      field = value
    }
}

fun main() {
  val lc = LogChanges()
  lc.n eq 0
  lc.n = 2
  lc.n eq 2
  trace eq "0 becomes 2"
}
```

Eğer bir özelliği `private` olarak tanımlarsanız, her iki erişimci de `private` olur. Ayrıca setter'ı `private` ve getter'ı `public` yapabilirsiniz. Bu durumda, özelliği sınıfın dışından okuyabilirsiniz, ancak değerini yalnızca sınıf içinde değiştirebilirsiniz:

```
// PropertyAccessors/Counter.kt
package propertyaccessors
import atomictest.eq

class Counter {
  var value: Int = 0
    private set
  fun inc() = value++
}

fun main() {
  val counter = Counter()
  repeat(10) {
    counter.inc()
  }
  counter.value eq 10
}
```

`private set` kullanarak, `value` özelliğini kontrol ediyoruz, böylece sadece bir artırılabilir.

Normal özellikler verilerini bir alanda saklar. Ayrıca alanı olmayan bir özellik de oluşturabilirsiniz:

```
// PropertyAccessors/Hamsters.kt
package propertyaccessors
import atomictest.eq

class Hamster(val name: String)

class Cage(private val maxCapacity: Int) {
  private val hamsters =
    mutableListOf<Hamster>()
  val capacity: Int
    get() = maxCapacity - hamsters.size
  val full: Boolean
    get() = hamsters.size == maxCapacity
  fun put(hamster: Hamster): Boolean =
    if (full)
      false
    else {
      hamsters += hamster
      true
    }
  fun take(): Hamster =
    hamsters.removeAt(0)
}

fun main() {
  val cage = Cage(2)
  cage.full eq false
  cage.capacity eq 2
  cage.put(Hamster("Alice")) eq true
  cage.put(Hamster("Bob")) eq true
  cage.full eq true
  cage.capacity eq 0
  cage.put(Hamster("Charlie")) eq false
  cage.take()
  cage.capacity eq 1
}
```

`capacity` ve `full` özellikleri herhangi bir altta yatan durumu içermez—her erişim anında hesaplanır. Hem `capacity` hem de `full`, fonksiyonlara benzer ve bunları bu şekilde tanımlayabilirsiniz:

```
// PropertyAccessors/Hamsters2.kt
package propertyaccessors

class Cage2(private val maxCapacity: Int) {
  private val hamsters =
    mutableListOf<Hamster>()
  fun capacity(): Int =
    maxCapacity - hamsters.size
  fun isFull(): Boolean =
    hamsters.size == maxCapacity
}
```

Bu durumda, özellikleri kullanmak okunabilirliği artırır çünkü kapasite ve doluluk kafesin özellikleridir. Ancak, tüm fonksiyonlarınızı özelliklere dönüştürmeyin—önce nasıl okunduklarına bakın.

- -

Kotlin stil rehberi, hesaplaması ucuz olduğunda ve nesne durumu değişmediği sürece her çağrıda aynı sonucu döndüren özellikleri fonksiyonlara tercih eder.

Özellik erişimcileri, özellikler için bir tür koruma sağlar. Birçok nesne yönelimli dil, bir özelliğe erişimi kontrol etmek için fiziksel bir alanı `private` yapmaya dayanır. Özellik erişimcileri ile bu erişimi kontrol etmek veya değiştirmek için kod ekleyebilirken, herkesin bir özelliği kullanmasına izin verebilirsiniz.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Özet 2

> Bu atom, Bölüm II'deki atomları Her Yerde Nesneler'den Özellik Erişimcileri'ne kadar özetler ve gözden geçirir.

Eğer deneyimli bir programcıysanız, bu atom Özet 1'den sonraki atomunuzdur ve bunu takiben atomları sırayla geçeceksiniz.

Yeni programcılar, bu atomu okumalı ve gözden geçirmek için alıştırmaları yapmalıdır. Buradaki bilgiler sizin için net değilse, o konunun atomunu tekrar inceleyin.

Konular, deneyimli programcılar için uygun bir sırada görünmektedir, bu da kitabın atomlarının sıralamasıyla aynı değildir. Örneğin, minimal test çerçevemizi atomun geri kalanında kullanabilmek için paketleri ve ithalatları tanıtarak başlıyoruz.

Paketler ve Test

Yeniden kullanılabilir herhangi sayıda kütüphane bileşeni, `package` anahtar kelimesi kullanılarak tek bir kütüphane adı altında toplanabilir:

```
// Summary2/ALibrary.kt
package com.yoururl.libraryname

// Components to reuse ...
fun f() = "result"
```

Birden fazla bileşeni tek bir dosyada toplayabilir veya bileşenleri aynı paket adı altında birden fazla dosyaya yayabilirsiniz. Burada `f()`'yi tek bileşen olarak tanımladık.

Bunu benzersiz kılmak için, paket adı geleneksel olarak ters alan adınızla başlar. Bu örnekte, alan adı `yoururl.com`.

Kotlin'de, paket adı içeriğinin bulunduğu dizinden bağımsız olabilir. Java, dizin yapısının tam nitelikli paket adına karşılık gelmesini gerektirir, bu nedenle `com.yoururl.libraryna`

paketi `com/yoururl/libraryname` dizini altında bulunmalıdır. Karışık Kotlin ve Java projeleri için, Kotlin'in stil kılavuzu aynı uygulamayı önerir. Saf Kotlin projeleri için, `libraryname` dizinini projenizin dizin yapısının en üst seviyesine yerleştirin.

Bir `import` ifadesi, bir veya daha fazla adı mevcut ad alanına getirir:

```
// Summary2/UseALibrary.kt
import com.yoururl.libraryname.*

fun main() {
  val x = f()
}
```

`libraryname`'den sonraki yıldız, Kotlin'in bir kütüphanenin tüm bileşenlerini içe aktarmasını söyler. Ayrıca bileşenleri tek tek seçebilirsiniz; detaylar Paketler bölümünde.

Bu kitabın geri kalanında, `main()` dışında fonksiyonlar, sınıflar vb. tanımlayan herhangi bir dosya için `package` ifadelerini kullanacağız. Bu, kitap içindeki diğer dosyalarla ad çakışmalarını önler. Sadece `main()` içeren bir dosyaya genellikle `package` ifadesi koymayacağız.

Bu kitap için önemli bir kütüphane, basit test çerçevemiz olan `atomictest`tir. `atomictest`, Ek A: AtomicTest bölümünde tanımlanmıştır, ancak kitabın bu noktasında anlamayacağınız dil özelliklerini kullanır.

`atomictest`'i içe aktardıktan sonra, `eq` (eşittir) ve `neq` (eşit değildir) ifadelerini neredeyse dil anahtar kelimeleriymiş gibi kullanırsınız:

```
// Summary2/UsingAtomicTest.kt
import atomictest.*

fun main() {
  val pi = 3.14
  val pie = "A round dessert"
  pi eq 3.14
  pie eq "A round dessert"
  pi neq pie
}
/* Output:
3.14
A round dessert
```

```
3.14
*/
```

eq/neq fonksiyonlarını herhangi bir nokta veya parantez olmadan kullanma yeteneğine *infix gösterimi* denir. `infix` fonksiyonlarını ya normal şekilde çağırabilirsiniz: `pi.eq(3.14)`, ya da infix gösterimini kullanarak: `pi eq 3.14`. Hem `eq` hem de `neq`, eq/neq ifadesinin sol tarafındaki sonucu gösteren ve sağ taraftaki ifadenin sola eşit olmadığında (veya `neq` durumunda eşit olduğunda) bir hata mesajı gösteren doğruluk iddialarıdır. Bu şekilde kaynak kodunda doğrulanmış sonuçları görürsünüz.

`atomictest.trace`, sonuçları eklemek için fonksiyon çağrı sözdizimini kullanır, bu sonuçlar daha sonra `eq` kullanılarak doğrulanabilir:

```
// Testing/UsingTrace.kt
import atomictest.*

fun main() {
  trace("Hello,")
  trace(47)
  trace("World!")
  trace eq """
    Hello,
    47
    World!
  """
}
```

`println()` fonksiyonunu etkili bir şekilde `trace()` ile değiştirebilirsiniz.

Her Yerde Nesneler

Kotlin, hem nesne yönelimli hem de fonksiyonel programlama paradigmalarını destekleyen bir *hibrit nesne-fonksiyonel* dildir.

Nesneler, verileri depolamak için `val` ve `var` içerir (bunlara *özellikler* denir) ve bir sınıf içinde tanımlanan fonksiyonları kullanarak işlemler gerçekleştirirler, bu fonksiyonlara *üye fonksiyonlar* denir (anlam karışıklığı olmadığında, sadece "fonksiyonlar" deriz). Bir *sınıf*, temelde yeni, kullanıcı tanımlı bir veri türü olan özellikleri ve üye

fonksiyonları tanımlar. Bir sınıfın `val` veya `var`'ını oluşturduğunuzda, buna *nesne oluşturmak* veya *örnek oluşturmak* denir.

Özellikle kullanışlı bir nesne türü *kapsayıcı*, başka bir adıyla *koleksiyon*dur. Kapsayıcı, diğer nesneleri tutan bir nesnedir. Bu kitapta, en genel amaçlı dizi olduğu için sık sık `List` kullanıyoruz. Burada `Double`'ları tutan bir `List` üzerinde birkaç işlem gerçekleştiriyoruz. `listOf()`, argümanlarından yeni bir `List` oluşturur:

```
// Summary2/ListCollection.kt
import atomictest.eq

fun main() {
  val lst = listOf(19.2, 88.3, 22.1)
  lst[1] eq 88.3  // Indexing
  lst.reversed() eq listOf(22.1, 88.3, 19.2)
  lst.sorted() eq listOf(19.2, 22.1, 88.3)
  lst.sum() eq 129.6
}
```

`List` kullanmak için herhangi bir `import` ifadesine gerek yoktur.

Kotlin, dizilere indeksleme yapmak için köşeli parantezler kullanır. İndeksleme sıfır tabanlıdır.

Bu örnek ayrıca `List`ler için mevcut olan birçok standart kütüphane fonksiyonundan birkaçını gösterir: `sorted()`, `reversed()` ve `sum()`. Bu fonksiyonları anlamak için çevrimiçi Kotlin belgelerini[28] inceleyin.

`sorted()` veya `reversed()` çağırdığınızda, `lst` değiştirilmez. Bunun yerine, istenen sonucu içeren yeni bir `List` oluşturulup döndürülür. Orijinal nesneyi asla değiştirmeme yaklaşımı Kotlin kütüphanelerinde tutarlıdır ve kendi kodunuzu yazarken bu deseni takip etmeye çalışmalısınız.

Sınıf Oluşturma

Bir sınıf tanımı, `class` anahtar kelimesi, sınıf için bir ad ve isteğe bağlı bir gövdeden oluşur. Gövde, özellik tanımlarını (`val` ve `var`) ve fonksiyon tanımlarını içerir.

Bu örnek, gövdesiz bir `NoBody` sınıfı ve `val` özelliklerine sahip sınıfları tanımlar:

[28] https://kotlinlang.org/docs/reference/

```
// Summary2/ClassBodies.kt
package summary2

class NoBody

class SomeBody {
  val name = "Janet Doe"
}

class EveryBody {
  val all = listOf(SomeBody(),
    SomeBody(), SomeBody())
}

fun main() {
  val nb = NoBody()
  val sb = SomeBody()
  val eb = EveryBody()
}
```

Bir sınıfın örneğini oluşturmak için, adının sonuna parantez koyun ve argümanlar gerekiyorsa onları da ekleyin.

Sınıf gövdelerindeki özellikler herhangi bir türde olabilir. `SomeBody`, `String` türünde bir özelliğe sahiptir ve `EveryBody`'nin özelliği `SomeBody` nesnelerini tutan bir `List`'tir.

İşte üye fonksiyonlara sahip bir sınıf:

```
// Summary2/Temperature.kt
package summary2
import atomictest.eq

class Temperature {
  var current = 0.0
  var scale = "f"
  fun setFahrenheit(now: Double) {
    current = now
    scale = "f"
  }
  fun setCelsius(now: Double) {
    current = now
```

```
    scale = "c"
  }
  fun getFahrenheit(): Double =
    if (scale == "f")
      current
    else
      current * 9.0 / 5.0 + 32.0
  fun getCelsius(): Double =
    if (scale == "c")
      current
    else
      (current - 32.0) * 5.0 / 9.0
}

fun main() {
  val temp = Temperature()   // [1]
  temp.setFahrenheit(98.6)
  temp.getFahrenheit() eq 98.6
  temp.getCelsius() eq 37.0
  temp.setCelsius(100.0)
  temp.getFahrenheit() eq 212.0
}
```

Bu üye fonksiyonlar, sınıfların *dışında* tanımladığımız üst düzey fonksiyonlar gibi, ancak sınıfa aittirler ve `current` ve `scale` gibi sınıfın diğer üyelerine niteliksiz erişime sahiptirler. Üye fonksiyonlar ayrıca aynı sınıftaki diğer üye fonksiyonları da niteliksiz çağırabilirler.

- **[1]** `temp` bir `val` olmasına rağmen, daha sonra `Temperature` nesnesini değiştiriyoruz. `val` tanımı, `temp` referansının yeni bir nesneye yeniden atanmasını engeller, ancak nesnenin davranışını kısıtlamaz.

Aşağıdaki iki sınıf bir tic-tac-toe oyununun temelini oluşturur:

```
// Summary2/TicTacToe.kt
package summary2
import atomictest.eq

class Cell {
  var entry = ' '                    // [1]
  fun setValue(e: Char): String =    // [2]
    if (entry == ' ' &&
      (e == 'X' || e == 'O')) {
      entry = e
      "Successful move"
    } else
      "Invalid move"
}

class Grid {
  val cells = listOf(
    listOf(Cell(), Cell(), Cell()),
    listOf(Cell(), Cell(), Cell()),
    listOf(Cell(), Cell(), Cell())
  )
  fun play(e: Char, x: Int, y: Int): String =
    if (x !in 0..2 || y !in 0..2)
      "Invalid move"
    else
      cells[x][y].setValue(e)        // [3]
}

fun main() {
  val grid = Grid()
  grid.play('X', 1, 1) eq "Successful move"
  grid.play('X', 1, 1) eq "Invalid move"
  grid.play('O', 1, 3) eq "Invalid move"
}
```

`Grid` sınıfı, her biri üç `Cell` içeren üç `List` içeren bir `List` barındırır—bir matris.

- **[1]** `Cell` içindeki `entry` özelliği bir `var` olarak belirlenmiştir, bu yüzden değiştirilebilir. Başlatma sırasında tek tırnak işaretleri bir `Char` türü oluşturur, bu nedenle `entry`'ye yapılan tüm atamalar da `Char` olmalıdır.
- **[2]** `setValue()` `Cell`'in müsait olup olmadığını ve doğru karakteri geçtiğinizi test eder. Başarı ya da başarısızlığı belirtmek için bir `String` sonucu döner.

- **[3]** `play()`, `x` ve `y` argümanlarının aralık içinde olup olmadığını kontrol eder, ardından matrise dizinler ve `setValue()` tarafından gerçekleştirilen testlere dayanır.

Yapıcılar

Yapıcılar yeni nesneler oluşturur. Bir yapıcıya, sınıf adının hemen ardından parantez içine yerleştirilen parametre listesi kullanılarak bilgi aktarılır. Bu nedenle bir yapıcı çağrısı, ilk harfinin büyük yazılması dışında (Kotlin stil kılavuzunu takip ederek) fonksiyon çağrısına benzer. Yapıcı, sınıfın bir nesnesini döner:

```
// Summary2/WildAnimals.kt
package summary2
import atomictest.eq

class Badger(id: String, years: Int) {
  val name = id
  val age = years
  override fun toString() =
    "Badger: $name, age: $age"
}

class Snake(
  var type: String,
  var length: Double
) {
  override fun toString() =
    "Snake: $type, length: $length"
}

class Moose(
  val age: Int,
  val height: Double
) {
  override fun toString() =
    "Moose, age: $age, height: $height"
}

fun main() {
```

```
  Badger("Bob", 11) eq "Badger: Bob, age: 11"
  Snake("Garden", 2.4) eq
    "Snake: Garden, length: 2.4"
  Moose(16, 7.2) eq
    "Moose, age: 16, height: 7.2"
}
```

`Badger`'daki `id` ve `years` parametreleri sadece *yapıcı gövdesinde* kullanılabilir. Yapıcı gövdesi, fonksiyon tanımlarının dışındaki kod satırlarından oluşur; bu durumda `name` ve `age` tanımlarıdır.

Çoğu zaman yapıcı parametrelerinin, yapıcı gövdesi dışındaki sınıfın diğer bölümlerinde de kullanılabilir olmasını istersiniz, ancak `name` ve `age` ile yaptığımız gibi yeni tanımlayıcılar tanımlama zahmetine girmeden. Parametrelerinizi `var` veya `val` olarak tanımlarsanız, bunlar özellik olur ve sınıfın her yerinde erişilebilir hale gelir. Hem `Snake` hem de `Moose` bu yaklaşımı kullanır ve yapıcı parametrelerinin artık ilgili `toString()` fonksiyonları içinde kullanılabilir olduğunu görebilirsiniz.

`val` ile ilan edilen yapıcı parametreleri değiştirilemez, ancak `var` ile ilan edilenler değiştirilebilir.

Bir nesneyi `String` bekleyen bir durumda kullandığınızda, Kotlin bu nesnenin `String` temsilini oluşturmak için onun `toString()` üye fonksiyonunu çağırır. Bir `toString()` tanımlamak için yeni bir anahtar kelimeyi anlamanız gerekir: `override`. Bu gereklidir (Kotlin bunu zorunlu kılar) çünkü `toString()` zaten tanımlanmıştır. `override`, Kotlin'e varsayılan `toString()`i kendi tanımımızla gerçekten değiştirmek istediğimizi söyler. `override`ın açıklığı, okuyucuya bunu net bir şekilde gösterir ve hataları önlemeye yardımcı olur.

`Snake` ve `Moose` için çok satırlı parametre listesinin formatına dikkat edin—bu, hem yapıcılar hem de fonksiyonlar için tek satıra sığmayacak kadar çok parametreye sahip olduğunuzda önerilen standarttır.

Görünürlüğü Sınırlama

Kotlin, C++ veya Java gibi diğer dillerde bulunanlara benzer *erişim belirleyicileri* sağlar. Bunlar, bileşen oluşturucularının müşteri programcıya nelerin sunulacağını belirlemesine olanak tanır. Kotlin'in erişim belirleyicileri `public`, `private`, `protected` ve `internal` anahtar kelimelerini içerir. `protected` daha sonra açıklanacaktır.

`public` veya `private` gibi bir erişim belirleyicisi, bir sınıf, fonksiyon veya özellik tanımından önce gelir. Her erişim belirleyici sadece o belirli tanımın erişimini kontrol eder.

`public` bir tanım herkes tarafından erişilebilir, özellikle de bu bileşeni kullanan müşteri programcı tarafından. Bu nedenle, `public` bir tanımdaki herhangi bir değişiklik müşteri kodunu etkileyecektir.

Eğer bir belirleyici belirtmezseniz, tanımınız otomatik olarak `public` olur. Belirli durumlarda netlik sağlamak için programcılar hala bazen gereksiz yere `public` belirtirler.

Bir sınıfı, üst seviye fonksiyonu veya özelliği `private` olarak tanımlarsanız, sadece o dosya içinde kullanılabilir:

```
// Summary2/Boxes.kt
package summary2
import atomictest.*

private var count = 0                    // [1]

private class Box(val dimension: Int) { // [2]
  fun volume() =
    dimension * dimension * dimension
  override fun toString() =
    "Box volume: ${volume()}"
}

private fun countBox(box: Box) {         // [3]
  trace("$box")
  count++
}

fun countBoxes() {
  countBox(Box(4))
  countBox(Box(5))
}

fun main() {
  countBoxes()
  trace("$count boxes")
  trace eq """
```

```
    Box volume: 64
    Box volume: 125
    2 boxes
  """
}
```

`private` özelliklere (**[1]**), sınıflara (**[2]**) ve fonksiyonlara (**[3]**) yalnızca `Boxes.kt` dosyasındaki diğer fonksiyonlar ve sınıflardan erişebilirsiniz. Kotlin, başka bir dosyadan `private` üst düzey öğelere erişmenizi engeller.

Sınıf üyeleri `private` olabilir:

```
// Summary2/JetPack.kt
package summary2
import atomictest.eq

class JetPack(
  private var fuel: Double   // [1]
) {
  private var warning = false
  private fun burn() =       // [2]
    if (fuel - 1 <= 0) {
      fuel = 0.0
      warning = true
    } else
      fuel -= 1
  public fun fly() = burn()  // [3]
  fun check() =              // [4]
    if (warning)             // [5]
      "Warning"
    else
      "OK"
}

fun main() {
  val jetPack = JetPack(3.0)
  while (jetPack.check() != "Warning") {
    jetPack.check() eq "OK"
    jetPack.fly()
  }
  jetPack.check() eq "Warning"
}
```

- **[1]** `fuel` ve `warning` her ikisi de `private` özelliklerdir ve `JetPack` üyeleri dışındaki kişiler tarafından kullanılamaz.
- **[2]** `burn()` `private`'tır ve bu nedenle yalnızca `JetPack` içinde erişilebilir.
- **[3]** `fly()` ve `check()` `public`'tir ve her yerde kullanılabilir.
- **[4]** Hiçbir erişim belirleyicisi olmaması, `public` görünürlük anlamına gelir.
- **[5]** Aynı sınıfın üyeleri yalnızca `private` üyelere erişebilir.

Bir `private` tanım herkes için *mevcut olmadığı* için, genellikle bunu müşteri programcısına aldırmadan değiştirebilirsiniz. Bir kütüphane tasarımcısı olarak, genellikle her şeyi mümkün olduğunca `private` tutarsınız ve yalnızca müşteri programcıların kullanmasını istediğiniz fonksiyonları ve sınıfları açarsınız. Bu kitapta örnek listelerin boyutunu ve karmaşıklığını sınırlamak için `private`'i yalnızca özel durumlarda kullanıyoruz.

Yalnızca *yardımcı bir fonksiyon* olduğuna emin olduğunuz herhangi bir fonksiyon `private` yapılabilir, bu şekilde yanlışlıkla başka bir yerde kullanmanızı ve dolayısıyla fonksiyonu değiştirme veya kaldırma yasağını önlemiş olursunuz.

Büyük programları *modüllere* bölmek faydalı olabilir. Bir modül, bir kod tabanının mantıksal olarak bağımsız bir parçasıdır. Bir `internal` tanım yalnızca tanımlandığı modül içinde erişilebilir. Bir projeyi modüllere nasıl böleceğiniz yapı sistemi (örneğin Gradle[29] veya Maven[30]) bağlıdır ve bu kitabın kapsamı dışındadır.

Modüller daha üst düzey bir kavramdır, oysa *paketler* daha ince taneli yapılandırmayı mümkün kılar.

İstisnalar

Bir `String`'i `Double`'a dönüştüren `toDouble()`'ı düşünün. Çevrilemeyen bir `String` için çağırırsanız ne olur?

[29] https://gradle.org/
[30] https://maven.apache.org/

```
// Summary2/ToDoubleException.kt

fun main() {
  // val i = "$1.9".toDouble()
}
```

`main()` fonksiyonundaki satırın yorumdan çıkarılması bir istisnaya yol açar. Burada, başarısız olan satır, kitabın derlemesini durdurmamak için yorumlanmıştır (bu kontrol her örneğin beklendiği gibi derlenip çalıştırıldığını kontrol eder).

Bir istisna fırlatıldığında, mevcut yürütme yolu durur ve istisna nesnesi mevcut bağlamdan çıkar. Bir istisna yakalanmadığında, program durur ve ayrıntılı bilgi içeren bir *yığın izi* görüntüler.

Kodları yorumlayarak veya yorumdan çıkararak istisnaların görüntülenmesini önlemek için, `atomictest.capture()` istisnayı saklar ve beklediğimizle karşılaştırır:

```
// Summary2/AtomicTestCapture.kt
import atomictest.*

fun main() {
  capture {
    "$1.9".toDouble()
  } eq "NumberFormatException: " +
    """For input string: "$1.9""""
}
```

`capture()` bu kitap için özel olarak tasarlanmıştır, böylece istisnayı görebilir ve çıktının kitabın yapı sistemi tarafından kontrol edildiğini bilebilirsiniz.

Fonksiyonunuz beklenen sonucu başarılı bir şekilde üretemediğinde başka bir strateji `null` döndürmektir. Daha sonra Null Değer Alabilen Tipler bölümünde `null`'un ifade türünü nasıl etkilediğini tartışacağız.

Bir istisna fırlatmak için, `throw` anahtar kelimesini ve ardından fırlatmak istediğiniz istisnayı, ayrıca gerekebilecek herhangi bir argümanı kullanın. Aşağıdaki örnekte `quadraticZeroes()` ikinci dereceden denklemi[31] çözer:

$$ax^2 + bx + c = 0$$

Çözüm *ikinci dereceden denklem*dir:

[31] https://en.wikipedia.org/wiki/Quadratic_formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

İkinci Dereceden Denklem

Örnek, parabolün x eksenini kestiği *kökleri* bulur. İki sınırlama için istisna fırlatıyoruz:

1. `a` sıfır olamaz.
2. Köklerin var olması için, **b² - 4ac** negatif olamaz.

Kökler varsa, iki tane vardır, bu yüzden dönüş değerlerini tutmak için `Roots` sınıfını oluşturuyoruz:

```
// Summary2/Quadratic.kt
package summary2
import kotlin.math.sqrt
import atomictest.*

class Roots(
  val root1: Double,
  val root2: Double
)

fun quadraticZeroes(
  a: Double,
  b: Double,
  c: Double
): Roots {
  if (a == 0.0)
    throw IllegalArgumentException(
      "a is zero")
  val underRadical = b * b - 4 * a * c
  if (underRadical < 0)
    throw IllegalArgumentException(
      "Negative underRadical: $underRadical")
  val squareRoot = sqrt(underRadical)
  val root1 = (-b - squareRoot) / (2 * a)
  val root2 = (-b + squareRoot) / (2 * a)
  return Roots(root1, root2)
}
```

```
fun main() {
  capture {
    quadraticZeroes(0.0, 4.0, 5.0)
  } eq "IllegalArgumentException: " +
    "a is zero"
  capture {
    quadraticZeroes(3.0, 4.0, 5.0)
  } eq "IllegalArgumentException: " +
    "Negative underRadical: -44.0"
  val roots = quadraticZeroes(1.0, 2.0, -8.0)
  roots.root1 eq -4.0
  roots.root2 eq 2.0
}
```

Burada standart istisna sınıfı `IllegalArgumentException` kullanıyoruz. Daha sonra kendi istisna türlerinizi tanımlamayı ve bunları kendi koşullarınıza özgü hale getirmeyi öğreneceksiniz. Amacınız, uygulamanızın gelecekteki desteğini basitleştirmek için mümkün olan en faydalı mesajları üretmektir.

Listeler

`Listler`, Kotlin'in temel sıralı kapsayıcı türüdür. `listOf()` kullanarak salt okunur bir liste ve `mutableListOf()` kullanarak değiştirilebilir bir liste oluşturursunuz:

```
// Summary2/ReadonlyVsMutableList.kt
import atomictest.*

fun main() {
  val ints = listOf(5, 13, 9)
  // ints.add(11) // 'add()' not available
  for (i in ints) {
    if (i > 10) {
      trace(i)
    }
  }
  val chars = mutableListOf('a', 'b', 'c')
  chars.add('d') // 'add()' available
  chars += 'e'
```

```
  trace(chars)
  trace eq """
    13
    [a, b, c, d, e]
  """
}
```

Temel bir `List` salt okunurdur ve değiştirme fonksiyonlarını içermez. Bu nedenle, `add()` değiştirme fonksiyonu `ints` ile çalışmaz.

`for` döngüleri `List` ile iyi çalışır: `for(i in ints)` ifadesi, `i`nin `ints` içindeki her bir değeri alması anlamına gelir.

`chars`, `MutableList` olarak oluşturulur; `add()` veya `remove()` gibi fonksiyonlar kullanılarak değiştirilebilir. Ayrıca, eleman eklemek veya çıkarmak için `+=` ve `-=` kullanabilirsiniz.

Salt okunur bir `List`, hiç değiştirilemeyen *değişmez* bir `List` ile aynı değildir. Burada, değiştirilebilir bir `List` olan `first`'i, salt okunur bir `List` referansı olan `second`'a atıyoruz. `second`'ın salt okunur özelliği, listenin `first` aracılığıyla değişmesini engellemez:

```
// Summary2/MultipleListReferences.kt
import atomictest.eq

fun main() {
  val first = mutableListOf(1)
  val second: List<Int> = first
  second eq listOf(1)
  first += 2
  // second sees the change:
  second eq listOf(1, 2)
}
```

`first` ve `second` bellekte aynı nesneye referans verir. `List`'i `first` referansı aracılığıyla değiştiririz ve sonra bu değişikliği `second` referansında gözlemleriz.

İşte üç tırnaklı bir paragrafı bölerek oluşturulan bir `String` `List`esi. Bu, bazı standart kütüphane fonksiyonlarının gücünü gösterir. Bu fonksiyonların nasıl zincirlenebildiğine dikkat edin:

```
// Summary2/ListOfStrings.kt
import atomictest.*

fun main() {
  val wocky = """
    Twas brillig, and the slithy toves
      Did gyre and gimble in the wabe:
    All mimsy were the borogoves,
      And the mome raths outgrabe.
  """.trim().split(Regex("\\W+"))
  trace(wocky.take(5))
  trace(wocky.slice(6..12))
  trace(wocky.slice(6..18 step 2))
  trace(wocky.sorted().takeLast(5))
  trace(wocky.sorted().distinct().takeLast(5))
  trace eq """
    [Twas, brillig, and, the, slithy]
    [Did, gyre, and, gimble, in, the, wabe]
    [Did, and, in, wabe, mimsy, the, And]
    [the, the, toves, wabe, were]
    [slithy, the, toves, wabe, were]
  """
}
```

`trim()`, baştaki ve sondaki boşluk karakterlerini (yeni satırlar dahil) kaldırarak yeni bir `String` üretir. `split()`, argümanına göre `String`'i böler. Bu durumda, bölünecek parçaları eşleştiren bir *düzenli ifade* oluşturan bir `Regex` nesnesi kullanıyoruz. `\W`, "kelime karakteri olmayan" anlamına gelen özel bir desendir ve `+`, "önceki karakterden bir veya daha fazlası" anlamına gelir. Bu nedenle, `split()`, bir veya daha fazla kelime olmayan karakterde bölünecek ve metin bloğunu bileşen kelimelerine ayıracaktır.

Bir `String` literalinde, `\`, özel bir karakterin önünde gelir ve örneğin, yeni bir satır karakteri (`\n`) veya bir sekme (`\t`) üretir. Sonuçta oluşan String'de gerçek bir `\` üretmek için iki ters eğik çizgi gereklidir: `"\\"`. Bu nedenle, tüm düzenli ifadeler, bir ters eğik çizgi eklemek için ekstra bir `\` gerektirir, üç tırnak işaretli `String` kullanmadığınız sürece: `"""\W+"""`.

`take(n)`, ilk `n` öğeyi içeren yeni bir `List` üretir. `slice()`, `Range` argümanıyla seçilen öğeleri içeren yeni bir `List` üretir ve bu `Range`, bir `adımı` içerebilir.

`sorted()` isminin `sort()` yerine kullanıldığına dikkat edin. `sorted()` çağrıldığında, orijinal `List`'i olduğu gibi bırakarak sıralı bir `List` *üretir*. `sort()` sadece bir `MutableList` ile çalışır ve bu liste *yerinde sıralanır*—orijinal `List` değiştirilir.

Adından da anlaşılacağı gibi, `takeLast(n)`, son `n` öğeden oluşan yeni bir `List` üretir. Çıktıdan "the"nin yinelendiğini görebilirsiniz. Bu, çağrı zincirine `distinct()` işlevi eklenerek ortadan kaldırılır.

Parametreli Türler

Tür parametreleri, en yaygın olarak kapsayıcılar olmak üzere bileşik türleri tanımlamamıza olanak tanır. Özellikle, tür parametreleri bir kapsayıcının ne içerdiğini belirtir. Burada, `numbers`'ın bir `Int` listesi, `strings`'in ise bir `String` listesi içerdiğini Kotlin'e bildiriyoruz:

```
// Summary2/ExplicitTyping.kt
package summary2
import atomictest.eq

fun main() {
  val numbers: List<Int> = listOf(1, 2, 3)
  val strings: List<String> =
    listOf("one", "two", "three")
  numbers eq "[1, 2, 3]"
  strings eq "[one, two, three]"
  toCharList("seven") eq "[s, e, v, e, n]"
}

fun toCharList(s: String): List<Char> =
  s.toList()
```

Hem `numbers` hem de `strings` tanımlamaları için, iki nokta üst üste ve `List<Int>` ve `List<String>` tip deklarasyonlarını ekliyoruz. Köşeli parantezler bir *tip parametresi* belirtir ve bu, "konteyner 'parametre' nesneleri tutar" dememize olanak tanır. `List<Int>` tipini tipik olarak "`List` of `Int`" olarak telaffuz edersiniz.

Bir dönüş değeri de bir tip parametresine sahip olabilir, `toCharList()`'te görüldüğü gibi. Sadece bir `List` döndüğünü söyleyemezsiniz—Kotlin şikayet eder, bu yüzden tip parametresini de vermelisiniz.

Değişken Argüman Listeleri

`vararg` anahtar kelimesi *değişken argüman listesi*nin kısaltmasıdır ve bir işlevin belirtilen tipte herhangi bir sayıda argümanı (sıfır dahil) kabul etmesine olanak tanır. `vararg`, `List`'e benzer bir `Array` olur:

```
// Summary2/VarArgs.kt
package summary2
import atomictest.*

fun varargs(s: String, vararg ints: Int) {
  for (i in ints) {
    trace("$i")
  }
  trace(s)
}

fun main() {
  varargs("primes", 5, 7, 11, 13, 17, 19, 23)
  trace eq "5 7 11 13 17 19 23 primes"
}
```

Bir fonksiyon tanımı yalnızca bir parametreyi `vararg` olarak belirtebilir. Listede herhangi bir parametre `vararg` olabilir, ancak sonuncusu genellikle en basitidir.

Bir `vararg` kabul edildiğinde, istediğiniz yere bir `Array` öğesi geçirebilirsiniz. Bir `Array` oluşturmak için, `arrayOf()`'u `listOf()`'u kullandığınız şekilde kullanın. Bir `Array` her zaman değiştirilebilir. Bir `Array`'i bir dizi argümana (yalnızca `Array` türünde tek bir öğe değil) dönüştürmek için, *yayılma operatörü* `*`'yi kullanın.

```
// Summary2/ArraySpread.kt
import summary2.varargs
import atomictest.trace

fun main() {
  val array = intArrayOf(4, 5)      // [1]
  varargs("x", 1, 2, 3, *array, 6)  // [2]
  val list = listOf(9, 10, 11)
  varargs(
    "y", 7, 8, *list.toIntArray())  // [3]
  trace eq "1 2 3 4 5 6 x 7 8 9 10 11 y"
}
```

Eğer yukarıdaki örnekte olduğu gibi ilkel tiplerden oluşan bir `Array` geçirirseniz, `Array` oluşturma fonksiyonu özel olarak yazılmalıdır. **[1]** `intArrayOf(4, 5)` yerine `arrayOf(4, 5)` kullanırsa, **[2]** bir hata üretir: *çıkarılan tip Array<Int> ancak IntArray bekleniyordu.*

Yayılma operatörü yalnızca dizilerle çalışır. Bir dizi argüman olarak bir `List` geçirmeniz gerekiyorsa, önce bunu bir `Array`'e dönüştürün ve ardından yayılma operatörünü uygulayın, **[3]**'te olduğu gibi. Sonuç bir ilkel tip `Array` olduğu için, özel dönüştürme fonksiyonu `toIntArray()` kullanmalıyız.

Kümeler

`Set`ler, her değerden yalnızca bir öğeye izin veren koleksiyonlardır. Bir `Set` otomatik olarak tekrarları önler.

```
// Summary2/ColorSet.kt
package summary2
import atomictest.eq

val colors =
  "Yellow Green Green Blue"
    .split(Regex("""\W+""")).sorted()  // [1]

fun main() {
  colors eq
    listOf("Blue", "Green", "Green", "Yellow")
```

```
  val colorSet = colors.toSet()        // [2]
  colorSet eq
    setOf("Yellow", "Green", "Blue")
  (colorSet + colorSet) eq colorSet    // [3]
  val mSet = colorSet.toMutableSet()   // [4]
  mSet -= "Blue"
  mSet += "Red"                        // [5]
  mSet eq
    setOf("Yellow", "Green", "Red")
  // Set membership:
  ("Green" in colorSet) eq true        // [6]
  colorSet.contains("Red") eq false
}
```

- **[1]** `String`, daha önce `ListOfStrings.kt` için açıklandığı gibi, bir regular expression kullanılarak `split()` edilir.
- **[2]** `colors`, salt okunur `Set` `colorSet` içine kopyalandığında, iki `"Green"` `String`'den biri yinelenen olduğu için kaldırılır.
- **[3]** Burada `+` operatörünü kullanarak yeni bir `Set` oluşturup görüntülüyoruz. Yinelenen öğeleri bir `Set` içine yerleştirmek, bu yinelenen öğeleri otomatik olarak kaldırır.
- **[4]** `toMutableSet()`, salt okunur bir `Set`'ten yeni bir `MutableSet` üretir.
- **[5]** Bir `MutableSet` için, `+ =` ve `-=` operatörleri, `MutableList`'lerde olduğu gibi, öğe ekler ve çıkarır.
- **[6]** `Set` üyeliğini `in` veya `contains()` kullanarak test edin.

Normal matematiksel küme işlemleri, birleşim, kesişim, fark vb. hepsi mevcuttur.

Mapler

Bir `Map`, *anahtarları değerlerle* ilişkilendirir ve bir anahtar verildiğinde bir değeri arar. `mapOf()`'a anahtar-değer çiftleri sağlayarak bir `Map` oluşturursunuz. `to` kullanarak, her anahtarı ilişkili değerinden ayırırız:

```
// Summary2/ASCIIMap.kt
import atomictest.eq

fun main() {
  val ascii = mapOf(
    "A" to 65,
    "B" to 66,
    "C" to 67,
    "I" to 73,
    "J" to 74,
    "K" to 75
  )
  ascii eq
    "{A =65, B =66, C =67, I =73, J =74, K =75}"
  ascii["B"] eq 66                    // [1]
  ascii.keys eq "[A, B, C, I, J, K]"
  ascii.values eq
    "[65, 66, 67, 73, 74, 75]"
  var kv = ""
  for (entry in ascii) {              // [2]
    kv += "${entry.key}:${entry.value},"
  }
  kv eq "A:65,B:66,C:67,I:73,J:74,K:75,"
  kv = ""
  for ((key, value) in ascii)         // [3]
    kv += "$key:$value,"
  kv eq "A:65,B:66,C:67,I:73,J:74,K:75,"
  val mutable = ascii.toMutableMap() // [4]
  mutable.remove("I")
  mutable eq
    "{A =65, B =66, C =67, J =74, K =75}"
  mutable.put("Z", 90)
  mutable eq
    "{A =65, B =66, C =67, J =74, K =75, Z =90}"
  mutable.clear()
  mutable["A"] = 100
  mutable eq "{A =100}"
}
```

- **[1]** Bir anahtar (`"B"`) `[]` operatörü ile bir değeri aramak için kullanılır. Tüm anahtarları `keys` ve tüm değerleri `values` kullanarak üretebilirsiniz. `keys`'e

erişmek bir `Set` üretir çünkü bir `Map` içindeki tüm anahtarlar zaten benzersiz olmalıdır (aksi takdirde arama sırasında karışıklık olurdu).
- **[2]** Bir `Map` içinde yineleme yapmak, harita girişleri olarak anahtar-değer çiftleri üretir.
- **[3]** Yineleme yaparken anahtar-değer çiftlerini açabilirsiniz.
- **[4]** Yalnızca okunabilir bir `Map`'ten `toMutableMap()` kullanarak bir `MutableMap` oluşturabilirsiniz. Şimdi `remove()`, `put()` ve `clear()` gibi değiştirme işlemleri yapabiliriz. Kare parantezler `mutable` içine yeni bir anahtar-değer çifti atayabilir. Ayrıca `map + = key to value` diyerek bir çift ekleyebilirsiniz.

Özellik Erişimcileri

Özellik `i`ye erişmek basit görünmektedir:

```
// Summary2/PropertyReadWrite.kt
package summary2
import atomictest.eq

class Holder(var i: Int)

fun main() {
  val holder = Holder(10)
  holder.i eq 10 // Read the 'i' property
  holder.i = 20  // Write to the 'i' property
}
```

Ancak, Kotlin okuma ve yazma işlemlerini gerçekleştirmek için fonksiyonlar çağırır. Bu fonksiyonların varsayılan davranışı, `i` içerisinde saklanan verileri okumak ve yazmaktır. *Özellik erişimcileri* yaratarak, okuma ve yazma işlemleri sırasında gerçekleşen eylemleri değiştirebilirsiniz.

Bir özelliğin değerini almak için kullanılan erişimciye *alıcı* denir. Kendi alıcınızı oluşturmak için, özellik tanımlamasının hemen ardından `get()` fonksiyonunu tanımlayın. Değişken bir özelliği değiştirmek için kullanılan erişimciye de *yazıcı* denir. Kendi yazıcınızı oluşturmak için, özellik tanımlamasının hemen ardından `set()` fonksiyonunu tanımlayın. Alıcıların ve yazıcıların tanımlanma sırası önemli değildir ve birini diğerinden bağımsız olarak tanımlayabilirsiniz.

Aşağıdaki örnekteki özellik erişimcileri, okuma ve yazma sırasında gerçekten çağrıldıklarını görebilmeniz için ek bilgi görüntülerken varsayılan uygulamaları taklit eder. `get()` ve `set()` fonksiyonlarını görsel olarak özellik ile ilişkilendirmek için girintili yapıyoruz, ancak gerçek ilişki bu fonksiyonların o özelliğin hemen ardından tanımlanmış olmalarından kaynaklanır:

```
// Summary2/GetterAndSetter.kt
package summary2
import atomictest.*

class GetterAndSetter {
  var i: Int = 0
    get() {
      trace("get()")
      return field
    }
    set(value) {
      trace("set($value)")
      field = value
    }
}

fun main() {
  val gs = GetterAndSetter()
  gs.i = 2
  trace(gs.i)
  trace eq """
    set(2)
    get()
    2
  """
}
```

Getter ve setter içinde, saklanan değer `field` anahtar kelimesi kullanılarak dolaylı olarak manipüle edilir, bu yalnızca bu iki fonksiyon içinde erişilebilir. Ayrıca sadece bir sonuç üretmek için getter'ı çağıran, ancak `field` içermeyen bir özellik de oluşturabilirsiniz.

Bir `private` özellik tanımlarsanız, her iki erişimci de `private` olur. Setter'ı `private` ve getter'ı `public` yapabilirsiniz. Bu, sınıfın dışından özelliği okuyabileceğiniz, ancak değerini yalnızca sınıf içinde değiştirebileceğiniz anlamına gelir.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Bölüm III: Kullanılabilirlik

Bilgisayar dilleri, neyi mümkün kıldıklarıyla değil, neyi kolaylaştırdıklarıyla farklılık gösterir—**Larry Wall**, Perl dilinin mucidi

Uzantı Fonksiyonları

> Diyelim ki ihtiyacınız olan her şeyi yapan bir kütüphane keşfettiniz... neredeyse. Sadece bir veya iki ek üye fonksiyonu olsaydı, sorunuzu mükemmel bir şekilde çözecekti.

Ama bu sizin kodunuz değil - ya kaynak koduna erişiminiz yok ya da onu kontrol etmiyorsunuz. Yeni bir sürüm çıktığında değişikliklerinizi tekrar etmek zorunda kalırdınız.

Kotlin'in *uzantı fonksiyonları*, mevcut sınıflara etkili bir şekilde üye fonksiyonlar ekler. Uzattığınız tipe *alıcı* denir. Bir uzantı fonksiyonu tanımlamak için, fonksiyon adının önüne alıcı tipini eklemelisiniz:

```
fun ReceiverType.extensionFunction() { ... }
```

Bu, `String` sınıfına iki uzantı fonksiyonu ekler:

```
// ExtensionFunctions/Quoting.kt
package extensionfunctions
import atomictest.eq

fun String.singleQuote() = "'$this'"
fun String.doubleQuote() = "\"$this\""

fun main() {
  "Hi".singleQuote() eq "'Hi'"
  "Hi".doubleQuote() eq "\"Hi\""
}
```

Uzantı fonksiyonlarını sanki sınıfın üyeleriymiş gibi çağırırsınız.

Başka bir paketten uzantıları kullanmak için, onları içe aktarmanız gerekir:

```
// ExtensionFunctions/Quote.kt
package other
import atomictest.eq
import extensionfunctions.doubleQuote
import extensionfunctions.singleQuote

fun main() {
  "Single".singleQuote() eq "'Single'"
  "Double".doubleQuote() eq "\"Double\""
}
```

Üye fonksiyonlara veya diğer uzantılara `this` anahtar kelimesini kullanarak erişebilirsiniz. `this`, bir sınıf içinde olduğu gibi burada da ihmal edilebilir, bu yüzden açık niteleme yapmanıza gerek yoktur:

```
// ExtensionFunctions/StrangeQuote.kt
package extensionfunctions
import atomictest.eq

// Apply two sets of single quotes:
fun String.strangeQuote() =
  this.singleQuote().singleQuote()  // [1]

fun String.tooManyQuotes() =
  doubleQuote().doubleQuote()       // [2]

fun main() {
  "Hi".strangeQuote() eq "''Hi''"
  "Hi".tooManyQuotes() eq "\"\"Hi\"\""
}
```

- **[1]** `this` `String` alıcısına atıfta bulunur.
- **[2]** İlk `doubleQuote()` fonksiyon çağrısının alıcı nesnesini (`this`) göz ardı ediyoruz.

Kendi sınıflarınıza uzantılar oluşturmak bazen daha basit kodlar üretebilir:

```
// ExtensionFunctions/BookExtensions.kt
package extensionfunctions
import atomictest.eq

class Book(val title: String)

fun Book.categorize(category: String) =
  """title: "$title", category: $category"""

fun main() {
  Book("Dracula").categorize("Vampire") eq
    """title: "Dracula", category: Vampire"""
}
```

Inside `categorize()` fonksiyonunun içinde, `title` özelliğine açık nitelik kullanmadan erişiyoruz.

- -

Genişletme fonksiyonları yalnızca genişletilen türün `public` (genel) öğelerine erişebilir. Bu nedenle, genişletmeler yalnızca normal fonksiyonların yapabileceği işlemleri gerçekleştirebilir. `Book.categorize(String)` fonksiyonunu `categorize(Book, String)` olarak yeniden yazabilirsiniz. Genişletme fonksiyonu kullanmanın tek nedeni sözdizimi olsa da, bu sözdizimi şekeri oldukça güçlüdür. Çağıran koda göre, genişletmeler üye fonksiyonlar gibi görünür ve IDE'ler, bir nesne için çağırabileceğiniz fonksiyonları listelerken genişletmeleri gösterir.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

İsimli & Varsayılan Argümanlar

Bir fonksiyon çağrısı sırasında argüman isimleri sağlayabilirsiniz.

İsimli argümanlar kod okunabilirliğini artırır. Bu, özellikle uzun ve karmaşık argüman listeleri için geçerlidir—isimli argümanlar, okuyucunun dokümantasyona bakmadan bir fonksiyon çağrısını anlamasını sağlayacak kadar açık olabilir.

Bu örnekte, tüm parametreler `Int`. İsimli argümanlar anlamlarını netleştirir:

```
// NamedAndDefaultArgs/NamedArguments.kt
package color1
import atomictest.eq

fun color(red: Int, green: Int, blue: Int) =
  "($red, $green, $blue)"

fun main() {
  color(1, 2, 3) eq "(1, 2, 3)"   // [1]
  color(
    red = 76,                     // [2]
    green = 89,
    blue = 0
  ) eq "(76, 89, 0)"
  color(52, 34, blue = 0) eq      // [3]
    "(52, 34, 0)"
}
```

- **[1]** Bu size pek bir şey söylemez. Argümanların ne anlama geldiğini bilmek için belgelere bakmanız gerekecek.
- **[2]** Her argümanın anlamı açıktır.
- **[3]** Tüm argümanları adlandırmanız gerekmez.

Adlandırılmış argümanlar, renklerin sırasını değiştirmenize olanak tanır. Burada, önce `blue` (maviyi) belirtiyoruz:

```
// NamedAndDefaultArgs/ArgumentOrder.kt
import color1.color
import atomictest.eq

fun main() {
  color(blue = 0, red = 99, green = 52) eq
    "(99, 52, 0)"
  color(red = 255, 255, 0) eq
    "(255, 255, 0)"
}
```

İsimli ve normal (konum) argümanları karıştırabilirsiniz. Argüman sırasını değiştirirseniz, çağrı boyunca isimli argümanlar kullanmalısınız—yalnızca okunabilirlik için değil, derleyiciye argümanların nerede olduğunu söylemek için de gereklidir.

İsimli argümanlar, *varsayılan argümanlar* ile birleştirildiğinde daha da kullanışlı hale gelir, bu argümanlar, fonksiyon tanımında belirtilen varsayılan değerlerdir:

```
// NamedAndDefaultArgs/Color2.kt
package color2
import atomictest.eq

fun color(
  red: Int = 0,
  green: Int = 0,
  blue: Int = 0,
) = "($red, $green, $blue)"

fun main() {
  color(139) eq "(139, 0, 0)"
  color(blue = 139) eq "(0, 0, 139)"
  color(255, 165) eq "(255, 165, 0)"
  color(red = 128, blue = 128) eq
    "(128, 0, 128)"
}
```

Sağlamadığınız herhangi bir argüman varsayılan değerini alır, bu yüzden sadece varsayılanlardan farklı olan argümanları sağlamanız yeterlidir. Uzun bir argüman listeniz varsa, bu kodunuzu basitleştirir ve yazmayı—daha da önemlisi—okumayı kolaylaştırır.

Bu örnek ayrıca `color()` tanımında *son eklenen virgül* kullanmaktadır. Son eklenen virgül, son parametreden (`blue`) sonraki ekstra virgüldür. Parametreleriniz veya değerleriniz birden fazla satırda yazıldığında bu kullanışlıdır. Son eklenen virgül ile yeni öğeler ekleyebilir ve bunların sırasını değiştirebilirsiniz, virgül ekleyip çıkarmadan.

Adlandırılmış ve varsayılan argümanlar (ve ayrıca son eklenen virgüller) kurucular için de çalışır:

```
// NamedAndDefaultArgs/Color3.kt
package color3
import atomictest.eq

class Color(
  val red: Int = 0,
  val green: Int = 0,
  val blue: Int = 0,
) {
  override fun toString() =
    "($red, $green, $blue)"
}

fun main() {
  Color(red = 77).toString() eq "(77, 0, 0)"
}
```

`joinToString()`, varsayılan argümanları kullanan bir standart kütüphane fonksiyonudur. Bir yinelenebilirin (bir liste, küme veya aralık) içeriğini bir `String`e dönüştürür. Bir ayraç, bir önek öğesi ve bir sonek öğesi belirtebilirsiniz:

```
// NamedAndDefaultArgs/CreateString.kt
import atomictest.eq

fun main() {
  val list = listOf(1, 2, 3,)
  list.toString() eq "[1, 2, 3]"
  list.joinToString() eq "1, 2, 3"
  list.joinToString(prefix = "(",
    postfix = ")") eq "(1, 2, 3)"
  list.joinToString(separator = ":") eq
    "1:2:3"
}
```

Varsayılan `toString()` bir `List` için içeriği köşeli parantezler içinde döndürür, ki bu istediğiniz şey olmayabilir. `joinToString()` parametrelerinin varsayılan değerleri `separator` için virgül ve `prefix` ve `postfix` için boş `String`lerdir. Yukarıdaki örnekte, sadece değiştirmek istediğimiz argümanları belirtmek için adlandırılmış ve varsayılan argümanları kullanırız.

`list` için başlatıcı bir virgülle sonlandırılır. Normalde, her bir öğe kendi satırında olduğunda bir sonlandırıcı virgül kullanırsınız.

Bir nesneyi varsayılan argüman olarak kullanırsanız, her çağrıda o nesnenin yeni bir örneği oluşturulur:

Eğer bir nesne örneğini varsayılan argüman olarak belirtirseniz (aşağıdaki örnekte `g()` içinde `da`), `g()` her çağrıldığında aynı örnek kullanılır. Eğer bir yapıcı çağrısı için sözdizimini (`h()` içinde `DefaultArg()`) belirtirseniz, `h()` her çağrıldığında o yapıcı çağrılır:

```
// NamedAndDefaultArgs/Evaluation.kt
package namedanddefault

class DefaultArg
val da = DefaultArg()

fun g(d: DefaultArg = da) = println(d)

fun h(d: DefaultArg = DefaultArg()) =
  println(d)

fun main() {
  g()
  g()
  h()
  h()
}
/* Sample output:
namedanddefault.DefaultArg@7440e464
namedanddefault.DefaultArg@7440e464
namedanddefault.DefaultArg@49476842
namedanddefault.DefaultArg@78308db1
*/
```

İki `g()` çağrısının çıktısı aynı nesne adreslerini gösterir. İki `h()` çağrısında ise,

`DefaultArg` nesnelerinin adresleri farklıdır, bu da iki ayrı nesne olduğunu gösterir.

Okunabilirliği artırdığında argüman isimlerini belirtin. Aşağıdaki iki `joinToString()` çağrısını karşılaştırın:

```
// NamedAndDefaultArgs/CreateString2.kt
import atomictest.eq

fun main() {
  val list = listOf(1, 2, 3)
  list.joinToString(". ", "", "!") eq
    "1. 2. 3!"
  list.joinToString(separator = ". ",
    postfix = "!") eq "1. 2. 3!"
}
```

`". "` veya `""` ayırıcı olup olmadığını tahmin etmek zor, parametre sırasını ezberlemediğiniz sürece ki bu pratik değil.

Varsayılan argümanlara başka bir örnek olarak, `trimMargin()` çok satırlı `String`leri biçimlendiren bir standart kütüphane fonksiyonudur. Her satırın başlangıcını belirlemek için bir kenar boşluğu öneki `String` kullanır. `trimMargin()`, her satırın başındaki boşluk karakterlerini ve ardından gelen kenar boşluğu önekini kırpar. İlk ve son satır boşsa, onları kaldırır:

```
// NamedAndDefaultArgs/TrimMargin.kt
import atomictest.eq

fun main() {
  val poem = """
    |->Last night I saw upon the stair
        |->A little man who wasn't there
          |->He wasn't there again today
|->Oh, how I wish he'd go away."""
  poem.trimMargin() eq
"""->Last night I saw upon the stair
->A little man who wasn't there
->He wasn't there again today
->Oh, how I wish he'd go away."""
  poem.trimMargin(marginPrefix = "|->") eq
"""Last night I saw upon the stair
A little man who wasn't there
```

```
He wasn't there again today
Oh, how I wish he'd go away."""
}
```

| ("pipe") varsayılan argümandır ve bunu kendi seçeceğiniz bir `String` ile değiştirebilirsiniz.

Alıştırmalar ve çözümler www.AtomicKotlin.com adresinde bulunabilir.

Aşırı Yükleme

> Varsayılan argüman desteği olmayan diller, bu özelliği taklit etmek için genellikle aşırı yüklemeyi kullanırlar.

Aşırı yükleme terimi, bir fonksiyonun adını ifade eder: Parametre listeleri farklı olduğu sürece, farklı fonksiyonlar için aynı adı (bu adı "aşırı yüklemek") kullanırsınız. Burada, `f()` üye fonksiyonunu aşırı yüklüyoruz:

```
// Overloading/Overloading.kt
package overloading
import atomictest.eq

class Overloading {
  fun f() = 0
  fun f(n: Int) = n + 2
}

fun main() {
  val o = Overloading()
  o.f() eq 0
  o.f(11) eq 13
}
```

`Aşırı yüklemede`, aynı isme sahip iki fonksiyon, `f()` görürsünüz. Fonksiyonun *imzası*, isim, parametre listesi ve dönüş türünden oluşur. Kotlin, imzaları karşılaştırarak bir fonksiyonu diğerinden ayırır. Fonksiyonları aşırı yüklerken, parametre listeleri benzersiz olmalıdır—dönüş türlerine göre aşırı yükleme yapamazsınız.

Çağrılar, gerçekten farklı fonksiyonlar olduklarını gösterir. Bir fonksiyon imzası, aynı zamanda kapsayıcı sınıf hakkında bilgi (veya genişletme fonksiyonu ise alıcı türü) içerir.

Eğer bir sınıf zaten aynı imzaya sahip bir üye fonksiyon içeriyorsa, Kotlin üye fonksiyonu tercih eder. Ancak, üye fonksiyonu bir genişletme fonksiyonu ile aşırı yükleyebilirsiniz:

```
// Overloading/MemberVsExtension.kt
package overloading
import atomictest.eq

class My {
  fun foo() = 0
}

fun My.foo() = 1              // [1]

fun My.foo(i: Int) = i + 2    // [2]

fun main() {
  My().foo() eq 0
  My().foo(1) eq 3
}
```

- **[1]** Bir üyenin kopyasını içeren bir uzantı bildirmek anlamsızdır, çünkü asla çağrılamaz.
- **[2]** Farklı bir parametre listesi sağlayarak bir uzantı fonksiyonu kullanarak bir üye fonksiyonunu aşırı yükleyebilirsiniz.

Varsayılan argümanları taklit etmek için aşırı yükleme kullanmayın. Yani, bunu yapmayın:

```
// Overloading/WithoutDefaultArguments.kt
package withoutdefaultarguments
import atomictest.eq

fun f(n: Int) = n + 373
fun f() = f(0)

fun main() {
  f() eq 373
}
```

Parametresiz fonksiyon sadece ilk fonksiyonu çağırır. İki fonksiyon, varsayılan bir argüman kullanılarak tek bir fonksiyonla değiştirilebilir:

```
// Overloading/WithDefaultArguments.kt
package withdefaultarguments
import atomictest.eq

fun f(n: Int = 0) = n + 373

fun main() {
  f() eq 373
}
```

Her iki örnekte de fonksiyonu ya bir argüman olmadan ya da bir tam sayı değeri geçerek çağırabilirsiniz. `WithDefaultArguments.kt` formunu tercih edin.

Varsayılan argümanlarla birlikte aşırı yüklenmiş fonksiyonları kullanırken, aşırı yüklenmiş fonksiyonu çağırmak "en yakın" eşleşmeyi arar. Aşağıdaki örnekte, `main()` içindeki `foo()` çağrısı, varsayılan argümanı olan 99'u kullanan ilk fonksiyon sürümünü *değil*, parametresiz olan ikinci sürümü çağırır:

```
// Overloading/OverloadedVsDefaultArg.kt
package overloadingvsdefaultargs
import atomictest.*

fun foo(n: Int = 99) = trace("foo-1-$n")

fun foo() {
  trace("foo-2")
  foo(14)
}

fun main() {
  foo()
  trace eq """
    foo-2
    foo-1-14
  """
}
```

`foo()` her zaman `f()` fonksiyonunun ikinci versiyonunu çağırdığı için, 99 varsayılan argümanını asla kullanamazsınız.

Aşırı yükleme neden kullanışlıdır? Farklı fonksiyon isimleri kullanmak zorunda olsaydınız, "bir tema üzerindeki varyasyonları" daha açık bir şekilde ifade etmenizi sağlar. Toplama fonksiyonları istediğinizi varsayalım:

```
// Overloading/OverloadingAdd.kt
package overloading
import atomictest.eq

fun addInt(i: Int, j: Int) = i + j
fun addDouble(i: Double, j: Double) = i + j

fun add(i: Int, j: Int) = i + j
fun add(i: Double, j: Double) = i + j

fun main() {
  addInt(5, 6) eq add(5, 6)
  addDouble(56.23, 44.77) eq
    add(56.23, 44.77)
}
```

`addInt()`, iki `Int` alır ve bir `Int` dönerken, `addDouble()`, iki `Double` alır ve bir `Double` döner. Aşırı yükleme olmadan, işlemi `add()` olarak adlandıramazsınız, bu yüzden programcılar genellikle benzersiz isimler üretmek için *ne* ile *nasıl*ı birleştirirler (rastgele karakterler kullanarak da benzersiz isimler oluşturabilirsiniz, ancak tipik desen, parametre türleri gibi anlamlı bilgileri kullanmaktır). Buna karşılık, aşırı yüklenmiş `add()` çok daha açıktır.

- -

Bir dilde aşırı yüklemenin olmaması büyük bir zorluk değildir, ancak bu özellik değerli bir basitleştirme sağlar ve daha okunabilir kod üretir. Aşırı yükleme ile sadece *neyi* söylersiniz, bu da soyutlama seviyesini yükseltir ve okuyucunun zihinsel yükünü azaltır. *Nasıl* olduğunu bilmek istiyorsanız, parametrelere bakın. Ayrıca, aşırı yüklemenin yinelenmeyi azalttığını da unutmayın: `addInt()` ve `addDouble()` dememiz gerekirse, esasen parametre bilgilerini fonksiyon adında tekrarlamış oluruz.

Alıştırmalar ve çözümleri www.AtomicKotlin.com adresinde bulunabilir.