2025

Android Geliştiricileri İçin Performans Rehberi

Murat Yüksektepe

v.1.0.1
https://muratyuksektepe.com

İçindekiler

1. mutable – immutable Karşılaştırması ... 8
2. Inline Fonksiyonlar ... 10
2.1. Noinline ... 12
3. Array ve List Karşılaştırması ... 12
3.1. Array ... 12
3.2. Collections (List, Set, Map) ... 12
3.3. Sonuç ... 13
4. Dizi Oluştururken Eleman Veri Türü Belirtmek ... 13
4.1. Tür Belirtilmemiş Diziler ... 13
4.2. Tür Belirtilmiş Diziler ... 15
4.3. Sonuç ... 16
5. Serializable – Parcelable Karşılaştırması ... 16
5.1. Serializable (java.io.Serializable) ... 17
5.2. Parcelable (android.os.Parcelable) ... 18
5.3. Sonuç ... 19
5.4. Ek Bilgi ... 19
6. Lateinit.isInitialized Yerine Nullable Değişkenler Kullanmak ... 21
6.1. if (::A.isInitialized) ... 22
6.2. if (A != null) ... 22
7. “by lazy {}” Kullanımı ... 23

Tanımlar

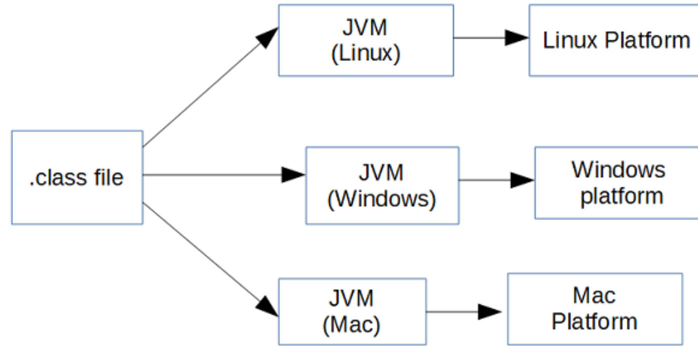
Compile-Time: Kaynak kodun, yürütülebilir bir koda dönüştüğü zaman.
Runtime: Yürütülebilir kodun çalışmaya başladığı zaman.

1. Sanal Makine (Virtual Machine)

Fiziksel cihazın kaynaklarını kullanarak o fiziksel cihaz üzerinde çalışan ve soyutlanmış bir alan yaratan yazılımlardır. Görevleri, belirli bir dilde yazılan kodları sanal makinenin derleyebileceği/çalıştırabileceği hâle getirmektir. Sanal makineler, aynı kodun farklı platform ve farklı donanıma sahip cihazlar üzerinde çalışabilmesine olanak tanırlar. Örneğin: Java Virtual Machine (JVM), farklı platformlar (Windows, Macintosh, Linux) üzerinde çalışabilir ve Java Bytecode'u çalıştırarak her platform için aynı çıktıyı üretir.

Android de APK dosyalarını çalıştırmak için sanal makine kullanır. Bunun birkaç büyük avantajı vardır:

- Uygulama, işletim sisteminden bağımsız ve izole olarak çalışır. Bu durumda uygulama kötü amaçlı kodlar içeriyor bile olsa işletim sistemi bundan etkilenmez. Bu izolasyon, Android işletim sisteminin güvenilir ve stabil bir hâlde kalmasını sağlar.
- Derleme ve derlenmiş kodun çalıştırılması aşamalarında platform bağımsız davranılabilir. Örneğin bilgisayarda derlenmiş bir kod aynı tip sanal makineyi kullanan mobil bir cihazda çalıştırılabilir.



Görsel 1: https://medium.com/programming-lite/android-core-jvm-dvm-art-jit-aot-855039a9a8fa

1.1. Dalvik Virtual Machine (DVM)

JVM; masaüstü bilgisayarlar ve sunucular gibi kaynak bakımından daha zengin cihazlar için geliştirilmişti; cep telefonları, tabletler, el terminalleri ve benzeri gibi kaynak bakımından daha kısıtlı cihazlar için uygun değildi. Bu sebeple Android cihazlarda JVM tercih edilmedi ve daha uygun bir çözüm olan DVM kullanılmaya başlandı ve Android 4.4 Kitkat sürümüne kadar varsayılan derleyici olarak kaldı. DVM'in JVM'e göre tercih edilmesinin birkaç önemli sebebi vardır.

- Yukarıda da bahsettiğimiz gibi DVM, mobil cihazlarda kullanılmak üzere geliştirilmiştir. Sınırlı işlemci ve bellek yönetimi için daha optimal çözümler sunar.
- *Just-In-Time* (JIT) derleme yöntemini kullanarak kodun sadece gerekli olan kısımlarını, uygulamanın çalışma zamanında derler.
- DVM, Android üzerinde çalışacak her uygulama için ayrı bir sanal makine başlatır. Bu özellik her uygulamanın birbirinden ve işletim sisteminden izole bir şekilde çalışmasına olanak tanır.
- JVM, *stack-based* bir yapıda çalışarak işlemleri bir yığın üzerinde gerçekleştirir. Buna mukabil olarak DVM ise *register-based* yapıda çalışarak mobil cihazlar gibi düşük bellek barındıran donanımlarda daha az kaynak tüketerek daha hızlı çalışabilirler.

1.2. Android Runtime (ART)

Android 4.4 KitKat ve sonraki sürümlerde deneysel bir özellik olarak tanıtılan ART, Android 5.0 Lollipop sürümüyle beraber varsayılan derleyici olarak kullanılmaya başlandı. ART'in DVM yerine tercih edilmesinin birkaç önemli sebebi vardır.

- DVM sadece JIT derleme yöntemini kullanırken ART ise hem JIT derleyicisini içebilir hem de asıl farkını ortaya çıkaran *Ahead-Of-Time* (AOT) derleyicisini kullanarak uygulamanın yüklenme zamanında tüm kodu derler ve elde ettiği makine kodunu cihazın belleğinde saklar. Bu işlem uygulamanın başlatılma süresini ve çalışma zamanında yaşanacak olan gecikmeleri ortadan

kaldırır, derleme işlemini sadece bir kez yaptığı için daha iyi performans ve daha az batarya kullanımı sağlar.

- Garbage Collector (GC) ‘da yapılan bazı iyileştirmeler bellek yönetimi açısından ART'ı, DVM'e göre daha performanslı hâle getirmiştir.
- ART, DVM'e göre daha az bellek kullanarak hem fiziksel cihaz üzerinde daha fazla bellek kalmasına hem de uygulamaların daha hızlı açılabilmesine olanak sağlar.
- ART, DVM'e göre daha fazla geliştirici aracı sağlar.

2. Bellek Yönetimi

Android uygulamaların bir sanal makine üzerinde çalıştığını öğrendikten sonra sanal makinelerin cihazın belleklerini nasıl kullandığını öğrenmemiz bellek yönetimi konusunda faydalı olacaktır.

Hem ART hem de Dalvik, tıpkı JVM'de olduğu gibi uygulamaları ve bu uygulamaların verilerini barındırmak için iki ayrı bellek alanı kullanırlar. Bu alanlar *stack* ve *heap* olarak adlandırılır.

2.1. Stack Memory (Yığın Bellek)

Stack memory, özellikle hızlı erişilmesi gereken ve nispeten az yer kaplayan veriler için kullanılan alandır. Özellikleri şu şekilde sıralanabilir:

- Bir fonksiyon çağırıldığında o fonksiyonun varlığı ve fonksiyon içinde yaratılan *local variables* (yerel değişkenler) bu alanda saklanır. (Bir fonksiyon içinde yaratılan değişkenlere diğer fonksiyonlardan ulaşamıyor olmamızın sebebi budur.)
- LIFO (Last In, First Out) yani “son giren ilk çıkar” prensibinde çalışır. Belleğe en son giren veri ilk önce geri alınır.
- Her fonksiyon çalıştığında verilerin saklanması için bir *stack frame* (yığın çerçevesi) oluşturulur ve fonksiyon görevini tamamladığında ilgili *stack frame*

bellekten silinir. (Fonksiyona ait yerel değişkenler ve fonksiyonun varlığı yok olur.)

- Çalışma prensibinin ve bellek yönetim şeklinin basitliği sebebiyle *stack memory* çok hızlıdır.
- Yukarıda da belirttiğimiz gibi *stack memory*, küçük ve hızlı erişilmesi gereken, ilkel türdeki (Int, Float, Char, Boolean vb gibi) veriler için idealdir.
- Cihazın donanımlarına ve işletim sisteminin ayarlarına göre değişmekle beraber her *thread* için ayrılan *stack memory* alanı 1 ilâ 8 MB arasında değişiklik gösterebilir. [1]
- Birbirini çok fazla kez veya sonsuz döngü ile çağıran fonksiyonlar çok fazla *stack frame* oluşturması nedeniyle *stack memory*'in dolmasına sebep olarak "StackOverflowError" hatası almamıza sebep olacaktır.

```
fun main() {
    a()
}

fun a() {
    a()
}
```

```
fun main() {
    fibonacci(100000)
}

fun fibonacci(position: Int): Int {
    return if (position <= 1) position
    else fibonacci(position - 1) +
fibonacci(position - 2)
}
```

Bu kodları çalıştırdığımızda "StackOverflowError" hatası alırız.

2.2. Heap Memory (Öbek Bellek)

Heap memory ise *stack memory*'nin aksine daha büyük veriler için kullanılan alandır. Özelliklerini şöyle sıralayabiliriz:

---

[1] https://www.b4x.com/android/forum/threads/solved-ble2-and-java-lang-stackoverflowerror-stack-size-8mb.159596/

- Birçok yerden erişilmesi için tanımladığımız *global variables* (global değişkenler) *heap memory* içinde barındırılır.
- İlkel veri türlerinin dışında kalan her nesne (class, object, model, list, …) *heap memory* içerisinde barındırılır.
- *Stack memory*'de sadece en son eklenmiş olan veriye ulaşabilirken *heap memory*'de böyle bir kısıtlama yoktur ve bu alan içerisinde yer alan herhangi bir veriye, o veriye ait referans bilgisi üzerinden istenilen zamanda erişilebilir.
- *Heap memory* içerisinde yer alan bir veriye ulaşmak *stack memory* ile karşılaştırıldığında biraz daha yavaştır.
- *Stack memory*'de işi biten veriler bellekten silinirken *heap memory*'de bu özellik yoktur. Kullanılmayan verilerin temizleme görevini *Garbage Collector* (GC) üstlenir.
- Android işletim sistemi, her uygulama için bir *heap memory* alanı ayırır. Bu alanın ne kadar olacağı cihazın türüne ve sahip olduğu RAM bellek boyutuna göre değişir.

```
val runtime = Runtime.getRuntime()
val maxMemory = runtime.maxMemory()
println("maxMemory: $maxMemory")
```

Bu kodu çalıştırarak ART'ın uygulamanız için ayırdığı heap alanını byte cinsinden görebilirsiniz.

- Eğer uygulamamız içerisinde çalışan bir kod blogu çok büyük boyutlu nesneler yaratıyorsa ve *heap memory* için ayrılan alanı aşıyorsak "OutOfMemoryError" hatası alırız.

```
val list = mutableListOf<String>()
while (true){
    val randomString = java.util.UUID.randomUUID().toString()
    list.add(randomString)
}
```

Bu kodu çalıştırdığımızda "OutOfMemoryError" hatasını alırız.

2.3. Garbage Collector (GC)

C ve C++ gibi düşük seviye programlama dillerinde bellek yönetiminin yazılımcı tarafından kontrol edilmesi ve hiçbir referansa sahip olmayan nesnelerin bellekten silinmesinin elle yapılması gerekmektedir. Bu durum zaman içinde sıkıcı bir hâl alabilir ve bellek yönetimi konusunun ihmal edilmesi uygulamamız açısından büyük sorunlara sebep olabilir.

Programlama dillerinin gelişmesiyle beraber hayatımıza giren *garbage collector* (çöp toplayıcısı) artık görevini tamamlamış, hiçbir referansa sahip olmayan nesnelerin bellekten silinme işlemini otomatik olarak yaparak biz yazılım geliştiricilere büyük kolaylık sağlamaktadır.

2.3.1. Android Runtime Garbage Collector

ART, bellek yönetimini en iyi şekilde yapabilmek için gelişmiş bir GC sistemi kullanır. Dalvik'ten ART'e geçme aşamasında daha iyi hâle getirilen bu GC sistemi optimal çözümler sunabilmek için farklı yapılar ve bu yapılar içinde farklı etkinlik ve özellikler barındırır. [2]

Concurrent Mark-Sweep (CMS) Algoritması

ART'den önceki sanal makine olan Dalvik'de var olan GC yapısıdır. Android KitKat sürümüne kadar varsayılan GC algoritması olarak kullanılmıştır.

Uygulama ile eş zamanlı olarak çalışan bu yapıda aslında sadece 2. adım olan "ulaşılabilir (direkt ya da dolaylı olarak bir referansa sahip) nesneleri işaretleme" ve 3. adım olan "ulaşılamayan (bir referansa sahip olmayan) nesnelerin silinmesi" işlemleri eş zamanlı olarak gerçekleşir. 1. adım olan "nesnelerin taranması" işlemi ise eş zamanlı değildir ve bu işlem yapılırken uygulama bir süre için kullanıcı tarafından kullanılamaz (bloklanmış) hâle gelir.

[2] https://source.android.com/docs/core/runtime/gc-debug#art_gc_overview

Concurrent Copying (CC) Algoritması

Android 8 (Oreo) sürümüyle birlikte ART'da yer alan GC yapısıdır ve bellek yönetimi açısından barındırdığı yeni özellikler ile performansı arttırmış ve maliyeti düşürmüştür.

CC, bellek yönetimini daha iyi hâle getirmek için bellek temizleme işleminden sonra *heap memory*'de oluşan boşlukları birleştirme mantığı içeren bir sıkıştırma yöntemi ***(heap compaction)*** kullanılır. Bu sayede:

- Bellek parçalanması (*memory fragmentation*) önlenir.
- Büyük boyutlu veriler için bellekte uygun alanlar açılmış olur.
- Bu sıkıştırma işlemi eş zamanlı olarak yapıldığı için uygulamayı bloklamaz.
- Google'a göre bu sıkıştırma yöntemi genel bellek ihtiyacı açısından 30% tasarruf sağlar.

Eğer uygulamamız içindeki bir *thread* 'in belleğe ihtiyacı olursa GC ona bütün sistemi ayırmak yerine kendi yerel nesnelerini barındırması için bir alan verir. İşlemler sonrası (sıkıştırma işlemi de eş zamanlı olarak devam ederken) bu alanda yer alan veriler 25% veya 30% gibi bir yer kaplıyorsa, uygun olan başka bir alana alınarak alan olası diğer ihtiyaçlar için boşaltılır. Bu yöntem sayesinde nesne oluşturma ve bellekte konumlandırma işlemleri eski Android sürümlerine göre çok daha hızlı gerçekleştirilir.

3. mutable – immutable Karşılaştırması

Kotlin'de *mutable* (değiştirilebilir) ve *immutable* (değiştirilemez, sabit) olmak üzere 2 tip değişken tanımlanabilir. `val` ile tanımladığımız değişkenler *immutable*'dır ve barındırdıkları veriler daha sonra değiştirilemezler. Bunun yanı sıra barındırdığı veriyi daha sonra değiştirmek istediğimiz değişkenkeri `var` ile tanımlarız.

`val` ile tanımlanan bir değişken için bytecode'a sadece `get` metodu oluşturulurken:

```
@NotNull
private static final String a = "A";

@NotNull
public static final String getA() {
   return a;
}
```

Kod 1: val a = "A" için oluşan Java kodu

var ile tanımlanan değişken için `get`'in yanı sıra bir de `set` metodu oluşturulur ve dinamik değişimi sağlamak için bellekte yedek alan bırakılır:

```
@NotNull
private static String b = "A";

@NotNull
public static final String getB() {
   return b;
}

public static final void setB(@NotNull String var0) {
   Intrinsics.checkNotNullParameter(var0, "<set-?>");
   b = var0;
}
```

Kod 2: var b = "B" için Java kodu

Bu durum göz önüne alınırsa hem Java kodunu azaltmak hem de belleği verimli kullanmak için gereksiz olan her yerde `var` yerine `val` ile değişken tanımlamak bizler için faydalı olacaktır. Android Studio'nun `var` ile tanımlanan bir değişken daha sonra değişime uğramamışsa bizlere `val`'e çevirmek konusunda uyarıda bulunmasının sebebi de budur.

Aynı şekilde bir dizi oluştururken de eğer daha sonra eleman ekleme işlemi yapılmayacaksa *mutable* diziler yerine *immutable* tipteki diziler tercih edilmelidir.

4.Inline Fonksiyonlar

Bildiğiniz üzere Kotlin dilinde geliştirme yaparken *high-order* fonksiyonlar oluşturup, bu fonksiyonlara parametre olarak başka fonksiyonlar (lambda) gönderebiliyoruz.

```
fun main() {
    val list = listOf(1, 2, 3, 4, 5)
    val random = Random.nextInt()
    list.each { println(random * it) }
}

fun <T> Collection<T>.each(block: (T) -> Unit) {
    for (e in this) block(e)
}
```

Kotlin'de kolayca yazdığımız bu yapı Java koduna çevrilirken, bizim alt fonksiyon olarak tanımladığımız parametre şeklindeki fonksiyonlar `new Function()` ile ayrıca yaratılır.

```
import kotlin.random.Random

new *
fun main() {
    val list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)
    val random = Random.nextInt()
    list.each { println(random * it) }
}

new *
fun <T> Collection<T>.each(block: (T) -> Unit) {
    for (e in this) block(e)
}
```

```
21   public static final void main() {
22      List list = CollectionsKt.listOf(new Integer[]{1, 2, 3, 4,
23      final int random = Random.Default.nextInt();
24      each((Collection)list, (Function1)(new Function1() {
25         // $FF: synthetic method
26         // $FF: bridge method
27         public Object invoke(Object var1) {
28            this.invoke(((Number)var1).intValue());
29            return Unit.INSTANCE;
30         }
31
           1 usage
32         public final void invoke(int it) {
33            int var2 = random * it;
34            System.out.println(var2);
35         }
36      }));
```

Şekil 1: Lambda fonksiyonun Java koda dönüşmüş hali

Örneğin çok uzun bir listenin her elemanı için çalıştıralacak bir alt fonksiyon gönderiyorsak, bu durum listenin eleman sayısı kadar yeni fonksiyon yaratılması ve bellekte ayrı ayrı yer kaplamasına sebep olacaktır. Şimdi bir de alt fonksiyonumuzun da

başka bir alt fonksiyon yarattığını düşünelim böylece bellekte yer kaplayacak olan yeni fonksiyon sayısı listenin eleman sayısının iki katına çıkmış olacaktır. Bellek yönetiminin optimal şekilde yapılandırılması göz önünde bulundurulduğunda bu kullanım pek istenmeyen bir kullanım şekli olacaktır. Buradaki sorun, sadece yeni fonksiyonların yaratılmasıyla ilgili değildir. Aynı zamanda bu yeni fonksiyonların çağırımları da işlem maliyetini arttıran bir süreçtir. Bu sorunun üstesinden gelmek için alt fonksiyonumuzu `inline` olarak tanımlamamız gerekir:

```
fun main() {
    val list = listOf(1, 2, 3, 4, 5)
    val random = Random.nextInt()
    list.each { println(random * it) }
}

inline fun <T> Collection<T>.each(block: (T) -> Unit) {
    for (e in this) block(e)
}
```

Bu sayede Kotlin kodumuz, Java'a dönüşürken parametre olarak gönderdiğimiz fonksiyonun içindeki kodlar ana fonksiyonumuzun içine taşınacak ve aldığımız sonuç değişmezken arkaplanda yaratılan gereksiz fonksiyon kalabalığından kurtulmuş oluruz.

Şekil 2: inline uygulanmış fonksiyon ve Java kodu

4.1. Noinline

Bir fonksiyonu `inline` ile işaretlediğimiz zaman parametre olarak aldığı tüm lambda fonksiyonların içeriğini ana fonksiyonun içerisine otomatik olarak taşıyacaktır. Şayet bazı lambda fonksiyonlar için bu işlemin gerçekleşmesini istemezsek (yani Java koduna çevirirken her biri için ayrı fonksiyon yaratılacak şekilde ayırsın istiyorsak) ilgili lambda fonksiyonlarını `noinline` ile işaretleyebiliriz.

```
fun main() {
    val list = listOf(1, 2, 3, 4, 5)
    val random = Random.nextInt()
    list.each { println(random * it) }
}

inline fun <T> Collection<T>.each(noinline block: (T) -> Unit) {
    for (e in this) block(e)
}
```

5. Array ve List Karşılaştırması

5.1. Array

Array'ler sabit eleman sayısını sağlamak için bitişik bellek konumlandırma yöntemini kullanırlar. Bu sayede doğrudan öğe erişimi ve öğe değişimi için daha az performansa ihtiyaç duyarlar. Fakat *Array* üzerinde yeni eleman ekleme, eleman çıkarma, sıralama, filtreleme gibi işlemler yapılırken var olan *Array* arka planda kopyalanarak üzerinde işlem yapılır, bu da istenmeyen ve verimsiz bir olaydır.

5.2. Collections (List, Set, Map)

Collection'larda ise bu işlemler dizi kopyalanmadan halledilir fakat dinamik yapılarını (büyüme ve küçülme) korumak için bir miktar fazladan bellek kullanımına ihtiyaç duyarlar.

5.3. Sonuç

Bu sebeple bir dizi üzerinde eleman ekme ve çıkarma işlemleri; sıralama (sort), ters çevirme (reverse), filtreme (filter), haritalama (map) vb. gibi işlemler çok az veya hiç yapılmayacaksa ve ekstra performans sağlamak amacı güdülüyorsa *List* yerine *Array* tercih edilmelidir.

6.Dizi Oluştururken Eleman Veri Türü Belirtmek

6.1. Tür Belirtilmemiş Diziler

Kotlin'de ister *Array* ister *Collection (List, Set, Map)* ile bir dizi oluştururken, dizinin içereceği tipi özel olarak belirtmememiz durumunda oluşturduğumuz dizi farklı türdeki verileri barındırabilir:

```
val list = listOf("A", 5, true, 'c', 3.14, 1f)
```

Böyle bir dizi için Kotlin Bytecode'u incelediğimiz de dizinin her bir elemanı için hangi türde olduğunu belirten bir `INVOKESTATIC` satırının var olduğunu görürüz. Bu durum Kotlin Bytecode'un uzamasına sebep olmaktadır:

```
    LINENUMBER 10 L0
    BIPUSH 6
    ANEWARRAY java/lang/Object
    DUP
    ICONST_0
    LDC "A"
    AASTORE
    DUP
    ICONST_1
    ICONST_5
    INVOKESTATIC java/lang/Integer.valueOf (I)Ljava/lang/Integer;
    AASTORE
    DUP
    ICONST_2
    ICONST_1
    INVOKESTATIC java/lang/Boolean.valueOf (Z)Ljava/lang/Boolean;
    AASTORE
    DUP
    ICONST_3
    BIPUSH 99
    INVOKESTATIC java/lang/Character.valueOf (C)Ljava/lang/Character;
    AASTORE
    DUP
    ICONST_4
    LDC 3.14
    INVOKESTATIC java/lang/Double.valueOf (D)Ljava/lang/Double;
    AASTORE
    DUP
    ICONST_5
    FCONST_1
    INVOKESTATIC java/lang/Float.valueOf (F)Ljava/lang/Float;
    AASTORE
    INVOKESTATIC kotlin/collections/CollectionsKt.listOf ([Ljava/lang/Object;)Ljava/util/List;
    ASTORE 0
   L1
```

Eleman tipi belirtilmemiş bir dizi içinde aynı tipte elemanların olması halinde de aynı durum geçerlidir. Örneğin `val list = listOf(5, 3, 4)` dizisi için oluşan Kotlin Bytecode şöyledir:

```
    LINENUMBER 10 L0
    ICONST_3
    ANEWARRAY java/lang/Integer
    DUP
    ICONST_0
    ICONST_5
    INVOKESTATIC java/lang/Integer.valueOf (I)Ljava/lang/Integer;
    AASTORE
    DUP
    ICONST_1
    ICONST_3
    INVOKESTATIC java/lang/Integer.valueOf (I)Ljava/lang/Integer;
    AASTORE
    DUP
    ICONST_2
    ICONST_4
    INVOKESTATIC java/lang/Integer.valueOf (I)Ljava/lang/Integer;
    AASTORE
    INVOKESTATIC kotlin/collections/CollectionsKt.listOf ([Ljava/lang/Object;)Ljava/util/List;
    ASTORE 0
   L1
```

6.2. Tür Belirtilmiş Diziler

Hâl böyleyken performansı arttırmak adına, bir dizi oluşturma esnasında eğer dizinin elemanları tek bir tür veriden oluşacaksa bu türün belirtilmesi, oluşan Kotlin Bytecode'u kısaltarak bizlere fayda sağlayacaktır. Yukarıda 5, 3 ve 4 (Int türünde) elemanları olan örnek dizimiz için şimdi türü belirtilmiş bir dizi oluşturalım. Yeni örneğimiz olan `val list = intListOf(5, 3, 4)` için oluşan Kotlin Bytecode şu şekilde olacaktır:

```
    LINENUMBER 11 L1
    ICONST_5
    ICONST_3
    ICONST_4
    INVOKESTATIC androidx/collection/IntListKt.intListOf
(III)Landroidx/collection/IntList;
    ASTORE 1
   L2
```

6.3. Sonuç

Eğer içerisinde barındıracağı veri tek bir tür ise, ister bir Array ister bir Collection (List, Set, Map) oluştururken bu türün belirtilmesi Kotlin Bytecode'u kısaltacağı için bellek kullanımı ve performans açısından bizlere yardımcı olacaktır.

7. Serializable – Parcelable Karşılaştırması

Android'de bir Activity'den diğerine veri gönderirken `Intent` sınıfını kullanırız ve `putExtra` fonksiyonu ile verilerimizi isimlendirerek oluşturduğumuz `Intent` objesine, hedef Activity'e taşıması için ekleriz. Ekleyeceğimiz verilerin tipleri `Int`, `Char` ve `Boolean` gibi ilkel veri türleri olabileği gibi `String`, `Array` gibi nispeten daha komplike veri türleri de olabilir. Bunun yanı sıra kendi oluşturduğumuz veri sınıflarını eklemek için izin verilen 2 yöntem vardır.

```
putExtra(String!, Bundle?) defined in android.content.Intent
putExtra(String!, Parcelable?) defined in android.content.Intent
putExtra(String!, Serializable?) defined in android.content.Intent
putExtra(String!, Array<(out) Parcelable!>?) defined in android.content.Intent
putExtra(String!, Array<(out) CharSequence!>?) defined in android.content.Intent
putExtra(String!, Array<(out) String!>?) defined in android.content.Intent
putExtra(String!, Boolean) defined in android.content.Intent
putExtra(String!, BooleanArray?) defined in android.content.Intent
putExtra(String!, Byte) defined in android.content.Intent
putExtra(String!, ByteArray?) defined in android.content.Intent
putExtra(String!, Char) defined in android.content.Intent
putExtra(String!, CharArray?) defined in android.content.Intent
putExtra(String!, CharSequence?) defined in android.content.Intent
putExtra(String!, Double) defined in android.content.Intent
putExtra(String!, DoubleArray?) defined in android.content.Intent
putExtra(String!, Float) defined in android.content.Intent
putExtra(String!, FloatArray?) defined in android.content.Intent
putExtra(String!, Int) defined in android.content.Intent
putExtra(String!, IntArray?) defined in android.content.Intent
putExtra(String!, Long) defined in android.content.Intent
putExtra(String!, LongArray?) defined in android.content.Intent
putExtra(String!, Short) defined in android.content.Intent
putExtra(String!, ShortArray?) defined in android.content.Intent
putExtra(String!, String?) defined in android.content.Intent
```

7.1. **Serializable** (java.io.Serializable)

- Android SDK'ten bağımsız olan Serializable, Java için hazırlanmış standart bir arayüzdür. Kısaca, Android uygulama geliştirme konunun dışında tutulsa bile, Java dili ile yazılan her projede kullanılabilir.
- İşaretleme tekniğini kullandığı için uygulaması çok basittir. Onunla işaretlediğimiz sınıflarımızın serileştirme işlemini Java kendisi halleder bu sebeple bizim bir çok kod satırı eklememize gerek kalmaz.

```
import java.io.Serializable

data class Person(val name: String, val age: Int) : Serializable
```

- Serializable, içinde yer alan verileri serileştirmek için *reflection* (yansıma) tekniğini kullanır. Bu sebeple arkaplanda bir çok geçici dosya oluşturulur. Bu oluşturulan dosyalar *garbage collector* (çöp toplayıcı) için ekstra iş yüküne sebep olur.

7.2. **Parcelable** (android.os.Parcelable)

- Serializable'a rakip olarak geliştirilen Parcelable ise Android SDK içerisinde yer alan bir arayüzdür ve sadece Android uygulama geliştirme için kullanılabilir.
- Parcelable, verileri serileştirmek için *reflection* (yansıma) tekniğini kullanmaz bu sebeple dosya oluşturma ve çöp toplama etkinliklerine sebep olmaz.
- Bunun yanı sıra Parcelable'ın uygulanması daha fazla kalıp kodun oluşmasına ve daha kalabalık bir görünüme sebep olur.

```
import android.os.Parcel
import android.os.Parcelable

data class Person(val name: String, val age: Int) : Parcelable {
    constructor(parcel: Parcel) : this(
        parcel.readString() ?: "",
        parcel.readInt()
    )

    override fun writeToParcel(parcel: Parcel, flags: Int) {
        parcel.writeString(name)
        parcel.writeInt(age)
    }

    override fun describeContents(): Int {
        return 0
    }

    companion object CREATOR : Parcelable.Creator<Parcelable> {
        override fun createFromParcel(parcel: Parcel): Parcelable {
            return Parcelable(parcel)
        }

        override fun newArray(size: Int): Array<Parcelable?> {
            return arrayOfNulls(size)
        }
    }
}
```

7.3. Sonuç

Sonuç olarak: Parcelable'ın, Serializable gibi yansıma tekniğini kullanmaması, *garbage collector*'a ek iş yükü çıkaracak geçici dosyalar oluşturmaması ve Android uygulama geliştirme için özel olarak tasarlanması onun en az rakibi kadar hatta bir çok test sonucuna göre daha hızlı çalıştığını kanıtlamaktadır. [3]

7.4. Ek Bilgi

Serializable ve Parcelable'i, Kotlinx plugin'ini ile kullanmak için:

- libs.verisom.toml

```
[versions]
…
kotlinPlugin = "2.0.20"

# =========================

[plugins]
…
kotlin-parcelize = { id = "org.jetbrains.kotlin.plugin.parcelize",
version.ref = "kotlinPlugin"  }

kotlin-serialization = { id = "org.jetbrains.kotlin.plugin.serialization",
version.ref = "kotlinPlugin"}
```

[3] https://www.developerphil.com/parcelable-vs-serializable/

- build.gradle.kts (project level)

```
plugins {
    alias(libs.plugins.android.application) apply false
    alias(libs.plugins.android.library) apply false
    alias(libs.plugins.jetbrains.kotlin.android) apply false
    …
    alias(libs.plugins.kotlin.parcelize) apply false
    alias(libs.plugins.kotlin.serialization) apply false
}
```

- build.gradle.kts (module level)

```
plugins {
    alias(libs.plugins.android.library)
    alias(libs.plugins.jetbrains.kotlin.android)
    …

    // Serializable
    alias(libs.plugins.kotlin.serialization)

    // Parcelable
    alias(libs.plugins.kotlin.parcelize)
}

android { … }

dependencies {
    …
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-
json:1.7.2")
}
```

- Person.kt (Serializable)

```
import kotlinx.serialization.Serializable

@Serializable
data class Person(val name: String, val age: Int)
```

- Person.kt (Parcelable)

```
import android.os.Parcelable
import kotlinx.parcelize.Parcelize

@Parcelize
data class Person(val name: String, val age: Int) : Parcelable
```

8. Lateinit.isInitialized Yerine Nullable Değişkenler Kullanmak

Bildiğiniz üzere Kotlin'de non-nullable (geçerisiz bırakılamaz) bir değişken tanımlarken varsayılan değerini de tanımlamamız gerekir. Ancak Context gerekliği, *Dependency Injection* (bağımlılık injeksiyonu)'nun kullanılması veya test fonksiyonlarının hazırlanması gibi bazı durumlarda; *non-null* (geçersiz bırakılamaz) bir değişkenin barındırması gereken veri, ilgili sınıfın yaratılmasından veya uzun sürecek bazı hesaplamalardan sonra elde edilebilir. Bu gibi durumlar için Kotlin'in sağladığı *lateinit modifier*'ını kullanabiliriz. `Lateinit` için kısaca: "geçersiz bırakılamaz bir değişken yaratıp barındıracağı veriyi sonradan atayacağımızın sözünü vermektir" diyebiliriz.

Eğer bu içerik atama işlemi senkron bir akışa dahil değilse yani `lateinit` ile işaretlediğimiz bir değişken üzerinde işlem yaparken, içine verinin atanıp atanmadığından emin değilsek; Kotlin Reflection'ı sayesinde değişkenimize ait `isInitialized` değerini kullanarak kontrol edebiliriz. Aksi takdirde söz verip içini doğru şekilde doldurmadığımız bir değişken ile işlem yapmaya çalıştığımız zaman UninitializedPropertyAccessException hatası alırız.

```
lateinit var name: String

fun main() {
    if (::name.isInitialized) {
        println(name)
    } else {
        println("İsim henüz tanımlanmadı!")
    }
}
```

8.1. if (::A.isInitialized)

Yukarıda bahsedilen `isInitialized`'ı kullanmak için Kotlin, üzerinde çalıştığımız nesnenin bir kopyasını yaratmak zorundadır. Bu da geçici de olsa o an için bellekte bir miktar fazla yer işgal etmek anlamına gelir. Yukarıda verilen örnek kod için *decompile* edilmiş Java dosyası **84 satır** olacaktır.

8.2. if (A != null)

Bunun yerine -mümkün olan yerler için- *nullable* değişkenler tanımlamış olsaydık:

```
var name: String? = null

fun main() {
    if (name != null) {
        println(name)
    } else {
        println("İsim henüz tanımlanmadı!")
    }
}
```

Örnek kodumuz bu şekilde olacaktır. *Decompile* edilmiş Java dosyamızı incelediğimiz zaman ise sadece **42 satır** olduğunu görürüz.

9.“by lazy {}” Kullanımı

`lateinit` için kısaca; “*non-null* bir değişkenin oluşturulmasının sonraki bir zamana bırakılması” şeklinde açıklama yapıyorken, `by lazy` için: “*non-null* ya da *nullable* olması fark etmeksizin bir değişkenin oluşturulmasının, o değişkenin ilk kez çağırılma zamanında yapılması“ diye tanımlayabiliriz. Kısa bir karşılaştırma yapmak gerekirse;

`by lazy`

- `val` ile tanımlanır.
- *Nullable* olabilir.
- İçereği verinin yaratılma esnasında atanması gerekir.
- İçerdiği veri sonradan değiştirilemez.
- Bir sorun olması durumunda *compile-time* sırasında hata vererek uygulamanın hatalı derlenmesini engeller.

```
fun main() {
    val a: String by lazy {
        "Bu bir metindir"
    }

    println(a)
}
```

`lateinit`

- `var` ile tanımlanır.
- *Nullable* olamaz.
- İçereceği verinin atanması sonraki bir zamana bırakılabilir.
- İçerdiği veri sonradan değiştirilebilir.
- Context gerekliliği, uzun süren hesaplamalar, *Dependency Injection* kullanılması veya test fonksiyonların hazırlanması gibi durumlarda tercih edilebilir.
- Bir sorun olması olması durumunda *runtime* sırasında UninitializedPropertyAccessException hatası verir ve uygulama kullanım esnasında *crash*’e sebep olur.

```
lateinit var b: String
fun main() {
    b = "Bu bir metindir"
    b = "Yeni metin"

    println(b)
}
```

Her iki seçenek de farklı durumlarda kullanılmak için kendine has özellikler ihtiva etmektedir. Bellek yönetiminin optimal şekilde yapılandırılması için uygun durumlarda `by lazy` kullanılması önerilmektedir. Fakat, `by lazy` ile ataması yapılmış bir değişkenin uzun bir zaman sonra veya hiç çağırılmaması durumunda, geçici bellekte barındırılan verinin bellek sızıntısına sebep olabileceği dikkat etmemiz gereken bir husus olarak belirtilebilir.