IntelliJ IDEA ile

AZ VE ÖZ
KOTLİN

Beyza KOYULMUŞ

Kotlin Nedir?

Kotlin programlama dili; Rusya merkezli yazılım şirketi, Jetbrain tarafından desteklenen Java sanal makinesi (JVM) üzerinde çalışan ve ayrıca Javascript kaynak koduna derlenebilir statik tipli bir programlama dilidir.

Statik ve Dinamik Tipli Programlama Dilleri

Type (Tip / Tür) :

- Veri tiplerini ve belirli bir programlama dilinde izin verilen manipülasyonu ifade eder.
- Verilerin yapısını ve bellekte nasıl depolandıklarını açıklar.
- Diziler, tam sayılar, boolean ve float her biri türdür.

Statik Tipli Programlama Dili Nedir?

Her değişken tipinin önceden belirtiliyor olmasıdır. Yani string bir değer tanımlıyorken başına string, sayı tanımlanırken int, double, float gibi tipleri yazıyoruz. Bu nedenle değişken tipleri program henüz çalışmıyorken bile bu tiplerin neler olduğunu biliyor. Bu da program henüz çalışmıyorken bile bir hata yapmışsanız sizi uyarır ve hatayı düzeltmenizi bekler.

C, C++, C#, Java, Scala, Haskel gibi diller statik programlama dilleridir.

Dinamik Programlama Dili Nedir?

Dinamik programlama dilleri ise statiğin aksine değişken tiplerinin programın çalışma anında belirlediği dillerdir. Yani ne string, ne int, ne double ne de bir array için herhangi bir değişken tipi belirtmenize gerek yoktur. Bazıları bunun geliştirme hızını artırdığını düşünür çünkü geliştiricinin hangi tiple olduğunu düşünmeden direk yazdığını savunsa da (haklılık payları yok değil) programın çalışma anına kadar herhangi bir hata var mı yok mu göremezler.

Lips, Perl, Ruby, Python, JavaScript gibi diller dinamik programlama dilleridir.

Kotlin' in Tarihçesi

Temmuz 2011' de JetBrains, JVM için yeni bir dil olan ve bir yıldır geliştirilmekte olan Project Kotlin' i duyurdu.

Şubat 2012' de JetBrains, projeyi Apache 2 lisansı altında açık kaynak olarak yayınlanmıştır.Kotlin v1.0, 15 Şubat 2016'da yayınlanmıştır. Bu ilk stabil sürümü olarak kabul edilir ve JetBrains bu sürümden itibaren geriye dönük uyumluluk taahhüdünde bulunmuştur.

Kotlin, Google I/O 2017'de resmi bir Android geliştirme dili olarak duyurulmuştur.

Kotlin' in Çıkış Amacı Nedir?

JetBrains lideri Dmitry Jemerov;

Scala dışındaki çoğu dilin aradıkları özelliklere sahip olmadığını söylemiştir. Ancak, Scala'nın düşük derleme zamanını apaçık bir eksiklik olarak gösterdi.

Kotlin'in belirtilen hedeflerinden biri, Java kadar çabuk derlemektir.

Kotlin ile Neler Yapılabilir?

JVM: Server – side (sunucu taraflı) tüm uygulamalar

Android: Android üzerinden çıkaracağınız uygulamalar

Browser: Javascript tabanlı web uygulamaları

Native: MacOS, İOS ve Gömülü sistem uygulamaları.

Niçin Kotlin Kullanmalıyız?

Kotlin ile javanın bir çok açığı kapatılmıştır.

Kotlin mi Java mı ?

Kotlin ile Gelen Yeniklerden Bazıları

- Data Class
- Top-level (packagelevel) sınıflar ve fonksiyonlar
- Var ve val kullanımı
- .equals() yerine == kullanımı
- get/set ön eklerinin kullanılmaması
- Sealed Sınıflar
- Tip kontrolü ve dönüşümleri (is ve as)
- Smart casts (akıllı dönüşümler)
- Safe (nullable) casting
- When Yapısı
- Extensions
- Delegates

Java To Kotlin Converter

En güzel özellik Java bir kodunuz varsa studio onu kotline çevirebiliyor.

Kotlin > Java

Kotlin daha dinamik daha enerjik bir dildir.

Örneğin Javada 20 satır ile yaptığımız, kotlinde iki satır olabiliyor.

Algoritma Kavramı

Bir problemin çözümlenmesinde kullanılan tüm yolların nasıl ilerlediğini gösteren yapılara Algoritma denir.

Bazı Popüler Algoritmalar

- SIRALAMA ALGORİTMALARI
- Selection Sort (Seçerek sıralama)
- Insertion Sort(Ekleme sıralaması)
- Bubbke Sort(Kabarcık sıralaması)
- Quick Sort(Hızlı Sıralama)
- FOURIER TRANSFORM and FAST FOURIER TRANSFORM ALGORİTMASI
- Binary Search(İkili arama)

…

Algoritma Analizi

Algoritma Analizi bilgisayar programlarının performans ve kaynak kullanımı üzerine teorik çalışmadır. Algoritma analizinde öncelikle ve özellikle performans üzerinde durulur. Bilgisayar Programlarında işlerin nasıl daha hızlı gerçekleştirilebileceği ele alınır.

Algoritma Performans Kriterleri Nelerdir?

Zaman(time)

Algoritma ne kadar hızlı performans gösteriyor?

Algoritmanın çalışma zamanını ne etkiler?

Algoritma için gerekli olan zaman nasıl tahmin edilebilir?

Algoritma için gerekli olan zaman nasıl azaltılabilir?

Alan(space)

Hangi veri yapısı ne kadar yer kaplar?

Ne tür veri yapıları kullanılabilir?

Veri yapısı seçimi çalışma zamanını nasıl etkiler?

Null Kavramı

- Null ifadesi değeri olmayan bir değerdir.
- Null bilgisayar sektöründe bir değer olarak kabul edilmektedir.
- Null “boş değer” **değildir**!
- Bilgisayarda boşluk bir değer olarak kullanılır. Null boşluktan farklı bir değerdir.

Null Kavramının Ortaya Çıkışı

Tony Hoare : Null Referansının Mucidi

Milyon Dolarlık Hata

Null referans kavramını hayatımıza 1965’te nesne yönelimli programlama ile uğraşırken sokan Tony Hoare;

2009 yılında bir geliştirmeyi yaparken tanımlanacak bütün referansların güvenli olmasını sağlamak ve implement etmesi çok kolay olduğu için null referansı yerleştiriyor.

Bunu da sonrasında yaşattığı streslere ve hüzünlere ithafen milyon dolarlık hata olarak tanımlıyor ve özürlerine ekliyor.

Ben buna milyar dolarlık hatam diyorum.

Bu, 1965'te boş referansın icadıydı. O zamanlar, nesneye yönelik bir dilde (ALGOL W) referanslar için ilk kapsamlı yazı sistemini tasarlıyordum.

Amacım, derleyici tarafından otomatik olarak gerçekleştirilen kontrollerle tüm referans kullanımlarının kesinlikle güvenli olmasını sağlamaktı. Ancak, **basitçe uygulanması çok kolay olduğu için boş bir referans** koymanın cazibesine karşı koyamadım.

Bu, **son kırk yılda muhtemelen bir milyar dolarlık acıya ve hasara neden olan sayısız hataya, kırılganlığa ve sistem çökmesine rol açtı.**

Son yıllarda, Microsoft'ta PREfix ve PREfast gibi bir dizi program çözümleyicisi referansları kontrol etmek ve bir risk varsa boş olmayabilecekleri konusunda uyarılar vermek için kullanılmıştır.

Spec # gibi daha yeni programlama dilleri, boş olmayan başvurular için bildirimler sunmuştur. **1965'te reddettiğim çözüm budur.**

- 3 parametre alan bir fonksiyonunuz var, ancak bunlardan 2 tanesi isteğe bağlı. Bu işlevi 2 parametre ile çağırırsanız, bu 3.parametreyi boş mu yoksa tanımsız mı yapar ?

  Cevap tanımsız!

İki değer arasındaki fark nedir?

Null, boş değerin her türün bir üyesi olduğu, değeri olmayan nötr bir davranışa sahip olacak şekilde tanımlanan bir nesnedir.

Tanımsız (Undefined), bildirilmiş ancak henüz atanmamış bir değişken veya değerin değerlendirilmesidir.

Null ile birlikte, programların çalışmalarda derlenmesine ve çökmesine izin verme, bellek sızıntılarına izin verme ve daha da kötüsü kodda güvenlik sorunlarına neden olma olasılığını açtı.

Dünya çapındaki şirketlere sızan bir bilgisayar solucanı olan CodeRed virüsü tüm ağları çökertti. İşe ve tüm sıradan bankacılığa, diğer işlere verilen kesintinin 4 milyar dolara mal olacağı tahmin ediliyordu.

Böyle büyük hataların önüne geçmek için Kotlin null değerini istenildiği gibi kullanamama şartı getirdi.

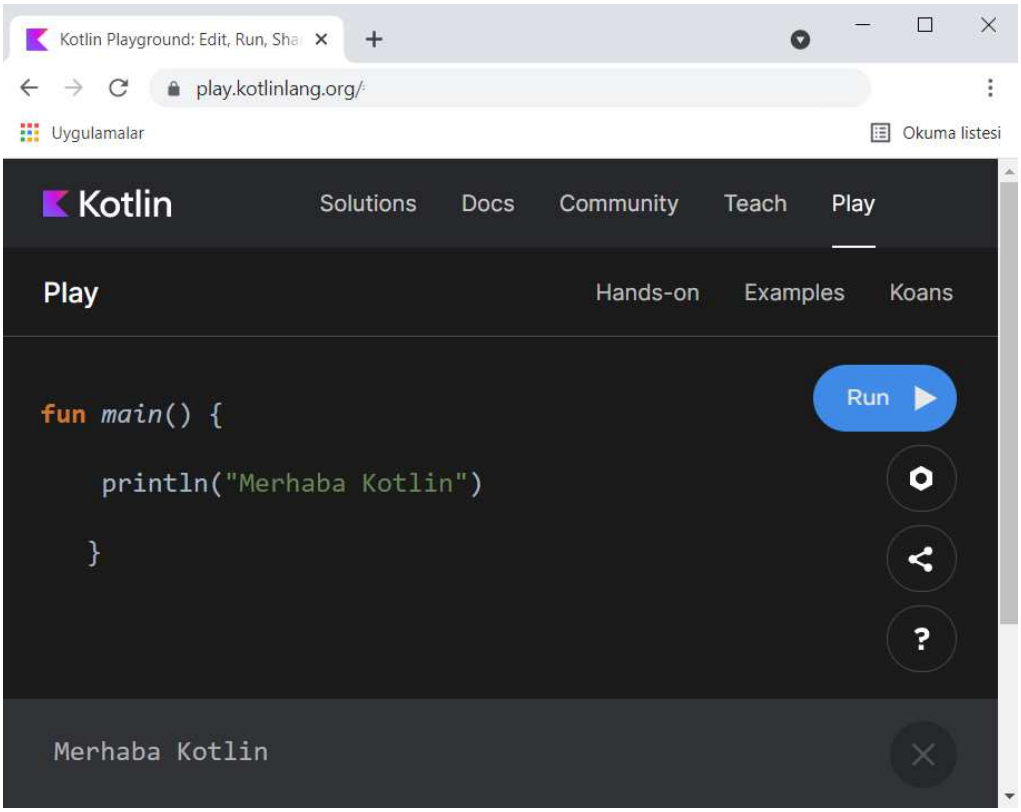
Kotlin Null-Safe'dir!

- Kotlin NullPointerException hatasının önüne geçmek için referansları null olabilir ve olamaz ayrımlarıyla bölünüyor.
- Kotlin'in bu güvenliği sağlayan özelliği tiplerin "non null" ya da "nullable" şeklinde tanımlanabilmesidir.
- Eğer ki " non null" bir değeri çağırmak isterseniz compiler hata verir. Bu da NullPointerException (NPE) hatasından korur.
- Örneğin String değişken tipi null değer alamaz.

Kotlin Geliştirme Ortamı

1. **Playgroud**'u kullanarak yani tarayıcı üzerinden online olarak veya
2. Bir **ide** kullanarak kotlin kodu yazabiliriz.

- Kotlin Playgroud'u kullanarak Kotlin kodunu çevrimiçi olarak çalıştırabiliriz. Bu site başlangıçta basit Kotlin kodlarını yazmak için Kotlin tarafından sağlanmıştır. Kotlin kodunu doğrudan tarayıcı üzerinde çalıştırabiliriz. Şimdilik online kotlin kodu yazmak için https://play.kotlinlang.org/ adresinden yazabiliriz.



  - Kodu çalıştırmak için Run butonuna basmamız yeterlidir.

- Ideyi anlatmadan ve ide üzerinde bir proje oluşturmadan önce birtakım kavramlardan bahsedeceğiz. Küçük örnekleri Kotlin Playgroud'u kullanarak yazacağız. Tarayıcı üzerinde çalışmak için: https://play.kotlinlang.org/

Merhaba Kotlin – İlk Kotlin Programı

İlk Kotlin programımızı yazalım. Merhaba Kotlin 😊 Bu basit programı Kotlin'de yazmak için gereken tüm unsurları inceleyeceğiz. Ancak doğrudan programımıza geçmeden önce, Fonksiyonlara bir göz atalım.

Fonksiyonlar Nelerdir?

Fonksiyonlar genel olarak bir şeyler yapan bir dizi talimattır, kotlin'de bir görevi gerçekleştiren bir kod bloğu.

- Örneğin bir pizzacıya gittiğimizde pizzada istediğimiz malzemeleri bildiririz. Bizim için istediğimiz pizza hazırlanır. Burada pizza dükkanı, girdilere (pizzanın türüne) dayalı olarak işlemler yapan ve bize çıktıyı, yani bizim istediğimiz pizzayı sağlayan bir fonksiyon gibi davranır.

Not: Fonksiyon yerine yöntem ve metot isimleri de kullanılmaktadır. Birbirlerinin yerlerine kullanılabilirler.

- Benzer şekilde, programlamada bir şeyler yapan bir metot tanımlarız. Programın yürütmesi için talimatlar yazarız ve bu işlevlere ne zaman ihtiyacımız olursa, bu metodu / fonksiyonu çağırırız.

Kotlin'de fonksiyon nasıl yazılır?

Fonksiyonun basit yapısı:	

	`fun fonksiyonIsmi(girdi) {`
	
	`// kod talimatları`
	
	`return çıktı`
	
	`}`

- Kotlin'de bu şekilde bir fonksiyon yazıyoruz. Şimdi ne zaman yazdığımız bu fonksiyona ihtiyaç duyarsak bu fonksiyonu kullanabileceğiz. Yani bu fonksiyonu çağıracağız, ona birtakım girdiler verdiğimizde sonucu/çıktıyı verecektir. Bunu ileriki kısımlarda daha detaylı inceleyeceğiz.

“Merhaba Kotlin“ Programı

- Programımızı çalıştırmaya başladığımızda çalışan özel bir fonksiyon vardır. Programımız bu fonksiyonu çağırarak çalışmaya başlar. Bu özel fonksiyon **Main Metodu (ana fonksiyon)** olarak bilinir.
- Birden fazla Kotlin dosyamız olduğunda JVM, main metodunun tanımlandığı bir dosya arar ve bu metodu yürütür. Bu yüzden main metodu yazmamız gerekiyor ki, bu programımızın giriş noktasıdır.

```
fun main()  {

      print("Merhaba Kotlin!")
      println("Merhaba Kotlin!")

        //Açıklama Satırı
        /**
         *Çoklu Açıklama Satırı
         */
}
```

```
Merhaba Kotlin!Merhaba Kotlin!

Process finished with exit code 0
```

- Programda ekrana Merhaba Kotlin! Merhaba Kotlin! Yazdırır.
- println çıktıyı yazdırır ve imleci bir sonraki satıra taşırken print sadece çıktıyı yazdırır.

Değişken Nedir?

- Değişken, verilerimizi sakladığımız bir kutudur. Bu verilere eriştiğimiz veya bu verileri güncellediğimiz bir adı vardır. Bir örnek verecek olursak, puanımızın olduğu bir oyun oynadığımızı düşünelim. Oyunda yaptığımız her bir işlemden sonra puan durumumuz değişir.
- Bu puan, uygulamamızda bir değişken olarak saklanır. Değişken, uygulamamız çalıştığında değişen verileri depolamak için bir yerdir. Bu yüzden **değişken** olarak adlandırılır. (Değişen bir şey)
- Özetle program içinde verilerle çalışmayı sağlayan sembolik ifadelerdir.

Kotlin’de Değişken Tanımlamak

- Kotlin’de 2 tür değişken vardır. Değişken tanımlamak için **var** ve **val (sabit)** anahtar kelimesi kullanılır.
  Aralarındaki fark şudur:
- Bir değişkene yeni bir değer atamak istersek onu **var** anahtar kelimesini kullanarak tanımlarız. Eğer değeri yeniden atamak istemiyorsak hep aynı değer kalacaksa **val** anahtar kelimesini kullanarak tanımlarız.

Örnek:

```
var skor=5

val oyuncuAdi="Beyza"
```

- İki değişken tanımlaması yaptık: skor, oyun ilerledikçe skoru güncellemeye devam etmek istediğimiz için **var** anahtar kelimesini kullanarak tanımlarken, oyuncu adı tüm oyun içinde aynı kaldığından dolayı oyuncuAdi değişkenini **val** kullanılarak tanımladık.
- Val değişkenine yeniden bir değer atanamaz. var, yeniden atanabilir.

```
var  instance : Tur = Deger
var adi : String ="Beyza"
var okullarTatilMi : Boolean = false
```

Değişken İsimlendirme Kuralları

- İlk karakter herhangi bir sayı veya rakamla başlayamaz.
- Değişken isimleri_(alt çizgi) ile başlayabilir.
- Değişken isimlerinde Türkçe karakter kullanmamalıyız.
- Değişken isimlerinde kelimeler arası boşluk kullanılmaz. Boşluk yerine bitişik kelimeler veya alt tire kullanabilirsiniz.
- Özel karakterler değişken isimlerinde kullanılmaz.
- Matematiksel işaretler ve programa ait komut ve deyimler değişken isimlerinde kullanılmaz.

Kotlin'de Sabit Tanımlamak

Uygulama içerisinde sahip olduğu değeri değiştirilemeyen yapılara **SABİT** denir.

- Değişkenler uygulamada farklı değerler alabilirken sabitler uygulamada **tek bir değere** sahip olabilir.
- Kotlin' de sabit tanımlamak için **val (value)** veya **const (constant)** anahtar ifadesi kullanılır. Java' da ise final anahtar ifadesi bu kullanıma denktir.

Örnek:

```
fun main() {
    val EKRAN_BOYUTU = getEkranBoyut()
    //Atama çalışma zamanında yapılır
}

//val, var gibi kullanılabiliyor.
var adi : String="Beyza"
val adi : String="Beyza" // adi ya yeni değer atanamaz.

companion object {
    const val EKRAN_BOYUTU=280
    //Atama derleme zamanında yapılır
}
```

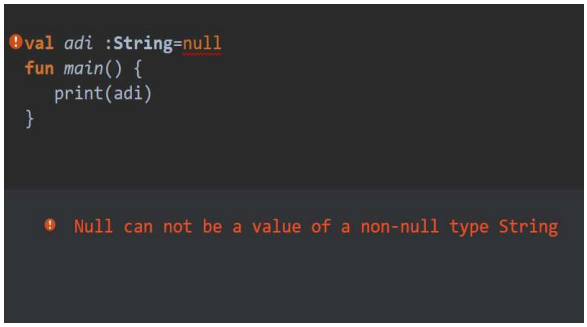
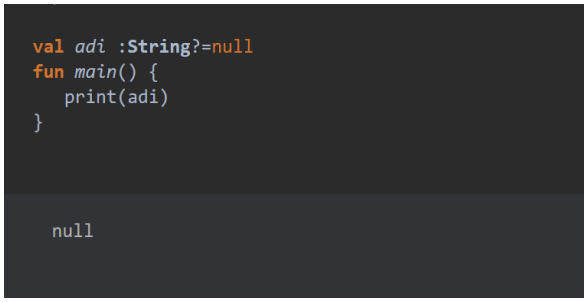
Kotlin'de Nullable Kullanımı

Kotlin programlama dilinde de null değeri kullanılır.

Farklı olarak Java'da olduğu gibi herhangi bir nesneye doğrudan null değeri verilmez.

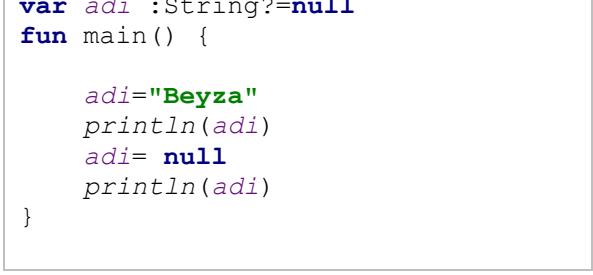
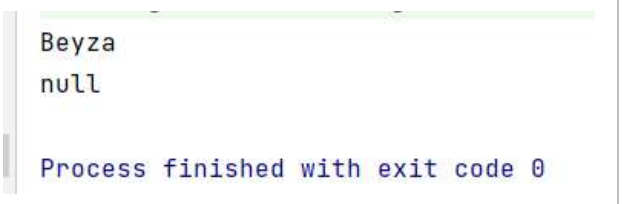
Bir nesna veya değişkenin null değerini alabilmesi için daha önceden nullable olarak belirlenmesi gerekmektedir.

- ? (Soru işareti) karakterini kullanarak o işlemi yapabiliriz.

	▪ Non-null olmayan bir objeye null atama yapamazsın hatasını verdi.
	▪ Null yapabilmek için, tür yanına soru işareti '?' koyarak null yapabileceğimizi söylüyoruz.

- ? karakterini kullanarak null değeri atayabiliyoruz. Ayrıca yeni bir değer ataması da yapılabiliyor.

Örnek:

Safe Call Kullanımı

Safe Call kullanımını bir örnek üzerinde anlatalım.

```
var name:String?=null
fun main() {
    println(name + "uzunluğu"+ name.length)

}
```

```
Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?
```

- Name değerini null yapıp boyutunu istediğimizde program hata verecektir. Peki dışarıdan null değeri gelince biz ne yapacağız?
- İşte burada ? sembolü kullanarak safe call yapmayı sağlayacağız. Boyutunu istediğimiz nesne sonuna ? koymamız yeterli olacaktır.

```
var name:String?=null
fun main() {
    println("name uzunluğu : "+ name?.length)

}
```

```
name uzunluğu : null
```

- name?.length yazarak name değeri doluysa yani null dan farklı bir şey ise çalışması gerektiğini bildirdik. Eğer dolu değilse null yazacaktır.

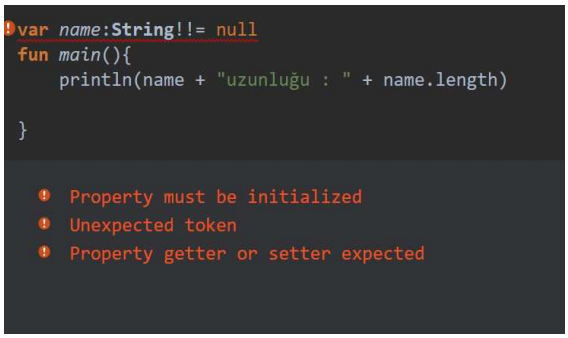
```
var name:String?="Beyza"
fun main() {
    println("name uzunluğu : "+ name?.length)

}
```

```
name uzunluğu : 5
```

Kotlin' de Non-Null Kullanımı

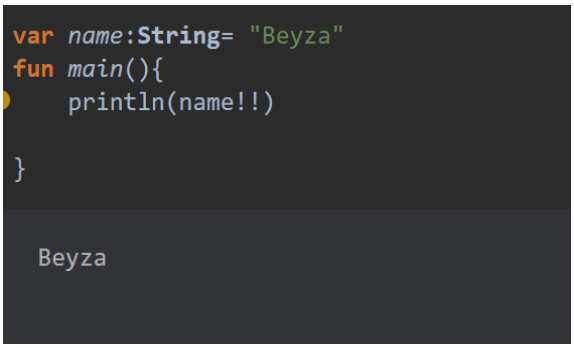
!! (Çift ünlem) karakterini kullanarak null bir değer gelmeyeceğini bildirebiliriz. Hiçbir şekilde null gelmeyeceğinin garanti olduğunu söylüyoruz. Daha iyi anlamak için örnekler üzerinde non-null kullanımını inceleyelim

Örnek:



- Yazdığımız kod hata verdi. Çünkü
- var name : String!! yazarak name değerinin asla null olmayacağını söyleyip, null değerini yazdığımız için hata verdi.

Örnek:



- name!! yazarak, asla null değeri gelmeyeceğinin garantisini vermiş olduk.

- Non – null kullanımı çok fazla tavsiye edilmez. Çünkü null değeri gelmeyeceğinin garantisini çoğu zaman veremeyiz. Eğer non – null kullanırsak dışarıdan null bir değer geldiğinde program patlayacaktır. Bu yüzden kullanımı önerilmez.

Kotlin' de Veri Tipleri

- Yukarıda belirttiğimiz gibi, değişkenler basit kutulardır. Şimdi bu kutuların **boyut**unu ve bu kutularda saklayacağımız **verilerin türünü** tanımlamamız gerekiyor.

1. **Number: Byte, Short, Int, Long, Float, Double**
2. **Boolean**
3. **Char**
4. **Strings**
5. **Arrays**

Type Inference (Tür Çıkarımı) Nedir?

Kotlin' de javanın aksine, bir değişken **tanımlarken tip belirtmek zorunlu değildir**. Kotlin, değişkene atanan değere göre tipi belirler.

Eğer bir değişkene kısa bir sayısal değer atanırsa bu değişken standart olarak Int tip alır. Uzun haneli bir sayısal değer için ise bu standart tip Long olur.

Örnek:

```
fun main()  {
    var yas= 60
    fun main(){

        println(yas)

    }

}
```

```
60
```

- yas tipini int olarak belirtmediğimiz halde bir hata ile karşılaşmadan 60 sonucunu aldık.

- Atadığımız değere göre kotlin , hangi türde bir değer atandığını biliyor.

Kotlin' de ve Javada Veri Tipleri Karşılaştırması

Java

- Java' da 8 tane hepsi küçük harf olan **İlkel (Primitive) Veri Tipleri** vardır.
  byte, short, long, integer, char, boolean, double, float

- **Referans Tipleri**
  Kendi referans tipimizi oluşturabiliyorduk
  Class, String, Byte, Integer, Interface, Array, Model…

Kotlin'de Primitive Tipler Yok Mu?

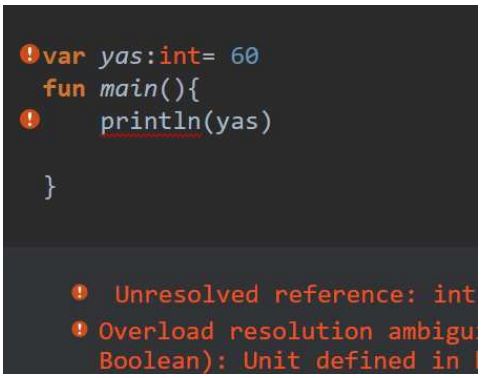
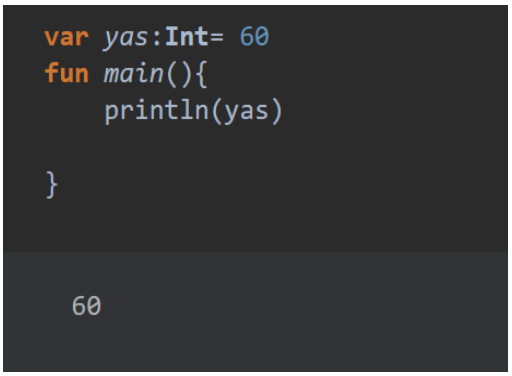
Kotlin dilinde primitive tipler yani int, float, double gibi küçük harfle başlayan veri tipleri yoktur ama aynı zamanda da vardır. Hem var hem yok 😊

Şöyle ki, Kotlin dilinde idemizin başına geçip kodlamaya başladığımızda primitive tipleri kullanamayız bunların yerine değerleri sarmalayan (wrapper) sınıflarını kullanırız.

Fakat Kotlin JVM tabanlı bir dil olduğu için bytecode çevirimi esnasında bizim kullandığımız değişkenleri primitive halinde düzenler. **Böylelikle primitive ve reference tipler arasındaki performans farkından etkilenmiyoruz.**

Yani işin özü, Kotlin syntaxında primitive tipler yoktur, bunun yerine sınıflar kullanılır fakat değişkenlerimiz JVM **tarafına primitive tip olarak** geçer.

Örnek:

int	Int
	

- int, primitive olarak kullandığımızda program hata verir.
- Referance tip olarak kullandığımızda bir hata ile karşılaşmayız.
- Sonuç olarak eğer tip belirtmek istersek, büyük harfle başlamalıdır.
- Int, String gibi.

Kotlin Number (Sayısal) Veri Tipleri

Kotlin'de ki sayısal veri tipleri **6 çeşittir**. Bunlar küçükten büyüğe doğru şunlardır:

- Byte
- Short
- Int
- Long
- Float
- Double

1a. Byte

- Byte veri türü bellekte 8 bitlik alan kaplar.
- Max alabileceği değer: 127, Min alabileceği değer: -128 dir.

1b. Short

- Short veri türü belleğimizde 16 bitlik alan kaplar.
- Max alabileceği değer: 32767, Min alabileceği değer: -32768

1c. Int

- Int veri türü bellekte 32 bitlik alan kaplar.
- Max alabileceği değer: 2 147 483 647, Min alabileceği değer: -2147483648

1d. Long

- Long veri türü bellekte 64 bitlik alan kaplar.
- Max alabileceği değer: $2^{63} - 1$, Min alabileceği değer: -2^{63}

1e. Float (Ondalıklı Sayı)

- Float veri türü bellekte 32 bitlik alan kaplar.
- (Max 7 basamaklı sayı)

1f. Double (Ondalıklı Sayı)

- Double veri türü bellekte 64 bitlik alan kaplar.
- (Max 16 basamaklı sayı)

Veri Türü	Kapladığı Alan	Max Değer	Min Değer
Byte	8 bit	127	-128
Short	16 bit	32 767	-32 768
Int	32 bit	2 147 483 647	-2 147 483 648
Long	64 bit	$2^{63} - 1$	-2^{63}
Float (Ondalıklı Sayı)	32 bit	...	...
Double (Ondalıklı Sayı)	64 bit	...	...

2. Boolean Veri Tipi

- Boolean veri tipi sadece iki değer alır, bunlar true ve false dir.

3. Char

- Kotlin'de **tek karakterli** veri saklamak için **Char** veri tipi kullanılır.
- **var** karakter: Char=**' B'**

4. String Veri Tipi

- String veri tipi içerisinde **metinsel** ,sayısal (ancak o da metinsel olarak algılanır yani sayısal işlem yapamazsınız) veriler tutabilir.
- **var** adi: String = **"Beyza"**

5. Array (Dizi) Veri Tipi

- Kotlin'deki diziler, farklı veri türlerinde olan değerleri bir arada tutabilen yapılardır. **(Not:** İleriki sayfalarda Array daha detaylı anlatılacaktır.**)**

IDE NEDİR?

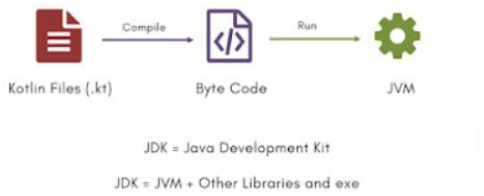
Kod yazabilmek için çeşitli araçlara ihtiyaç duyarız. Bu konuda ideler bize yardımcı olur. Programlama işlemlerini organize ederler. Birçok eklentiyle geliştirme işlemlerini daha kolay bir şekilde halletmemizi sağlarlar. IDE, Integrated Development Environment (Tümleşik Geliştirme Ortamı) kısaltmasıdır. IDE'ler daha verimli ve düzenli olma olanağı sunar. Ayrıca kodun otomatik tamamlama, debugging ve versiyon kontrolü yapabilme olanağı tanırlar. Kod dosyalarının hiyerarşik olarak görüntüleyebilme imkanı da sunar.

IDE programları, birbirinden farklı özelliklere sahiptir. Arayüz ve desteklediği dil açısından değişiklik gösterebilirler. İçerisinde barındırdıkları kütüphaneler sayesinde daha rahat kod yazabilmemize yardımcı olur. Her biri farklı iş akışına sahip olduğundan kendimize uygun olan ideyi seçmemiz verimlilik açısından önemlidir.

Kotlin kodunu yazmamız için kullanabileceğimiz birkaç ide mevcuttur. Biz IntelliJ Idea'yı kullanacağız. Ancak ideyi kurmadan önce bir JDK'ya ihtiyacımız var.

Kotlin ile çalışırken neden bir Java JDK' ya ihtiyacımız var?

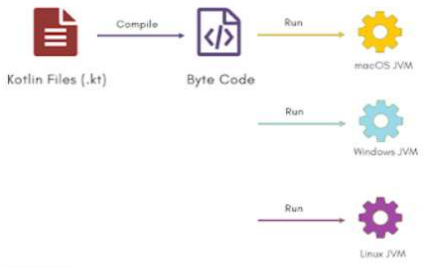
Kotlin bir JVM dilidir. Kotlin kodu, JVM üzerinde çalışacak olan bayt koduna derlenir. Bu yüzden bir JVM' ye ihtiyacımız var. Bununla birlikte, Kotlin programını çalıştırmak için JDK'da bulunan birkaç Java kitaplığına ihtiyacımız var. Java ve Kotlin birlikte çalışabilir. Java kodunu Kotlinden ve Kotlin kodunu Java'dan ayarlayabiliriz.



.kt, Kotlin kaynak kodu dosyalarının uzantısıdır.

JDK ve JVM?

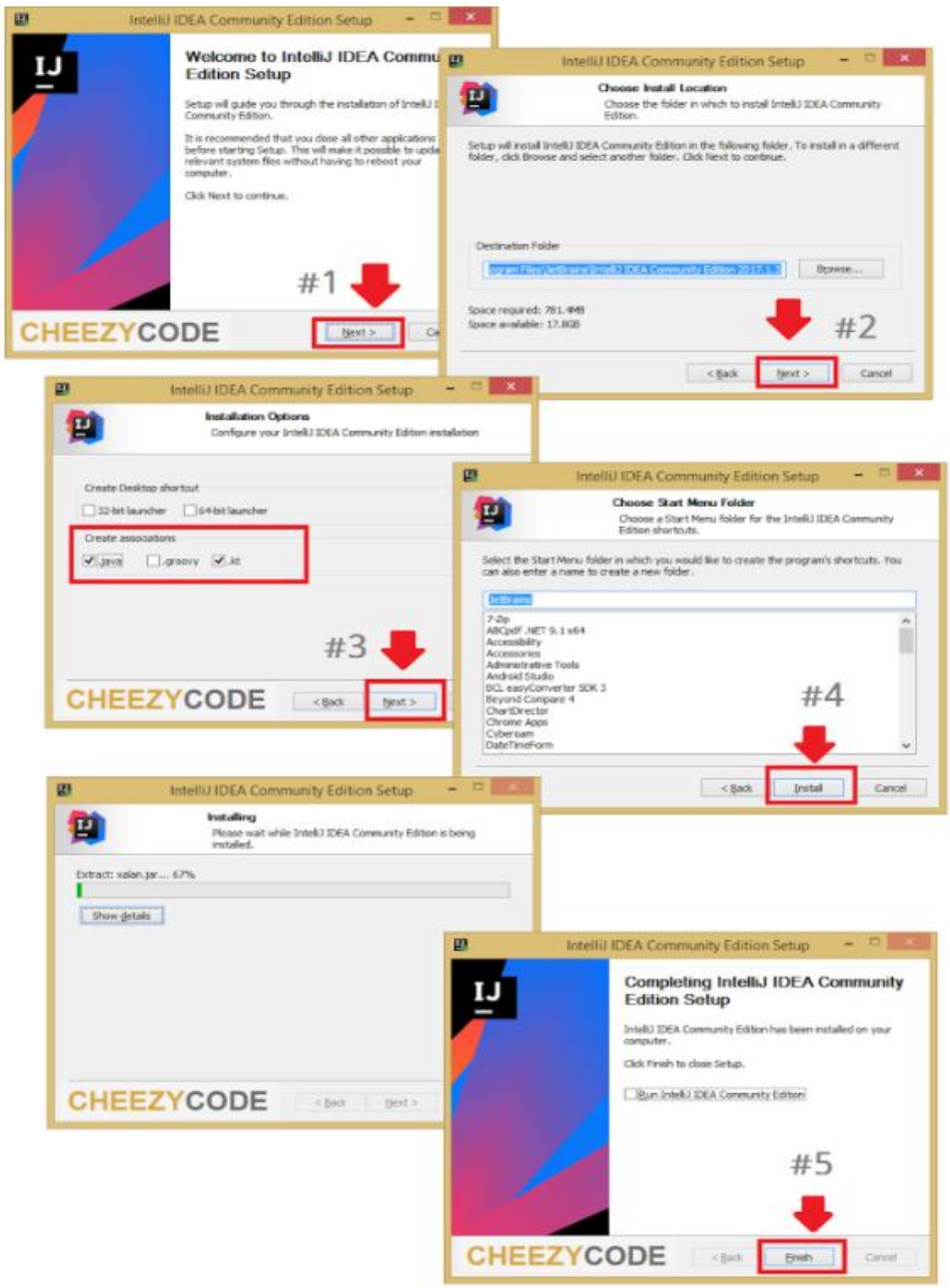
JVM bir Java Sanal Makinesidir. Kotlin kodunu derlediğimizde, bytecode'a dönüştürülüdür. Bu bayt kodu JVM üzerinde çalışır. Bu bayt kodu oluşturma mekanizması nedeniyle, Kotlin ve Java kodu farklı ortamlarda çalışabilir. Belirli bir ortam için bir JVM'ye ihtiyacımız var.



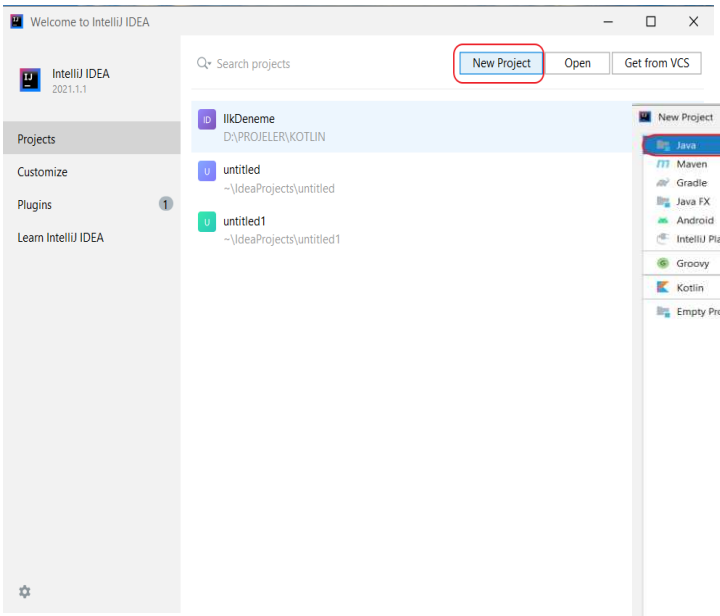
IDE ile Kotlin Çalışma Ortamı

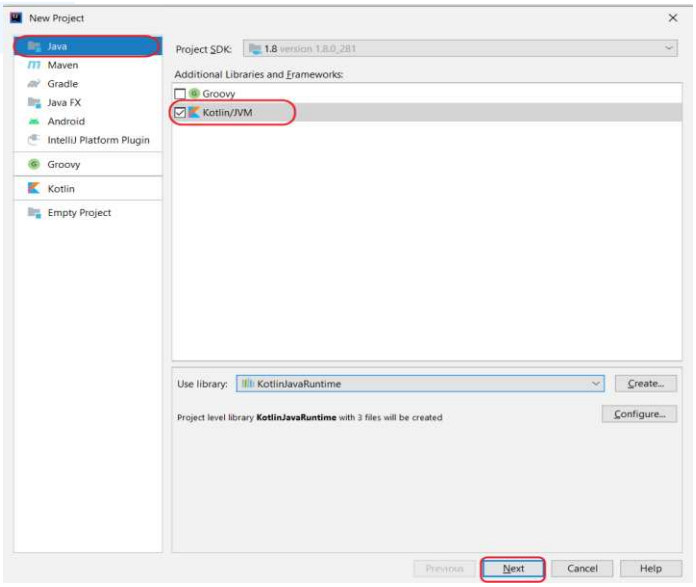
1. **JDK yükleme**
    - https://www.oracle.com/tr/java/technologies/javase-downloads.html linkinden ulaşabiliriz.
    - Java SE 8 indirmemiz ideal olacaktır. Eğer bilgisayarınızda mevcut bir JDK kurulu ise tekrar yüklemenize gerek yoktur. Cihazınızda JDK olup olmadığını CMD kısmına" java -version" yazarak kontrol edebilirsiniz.

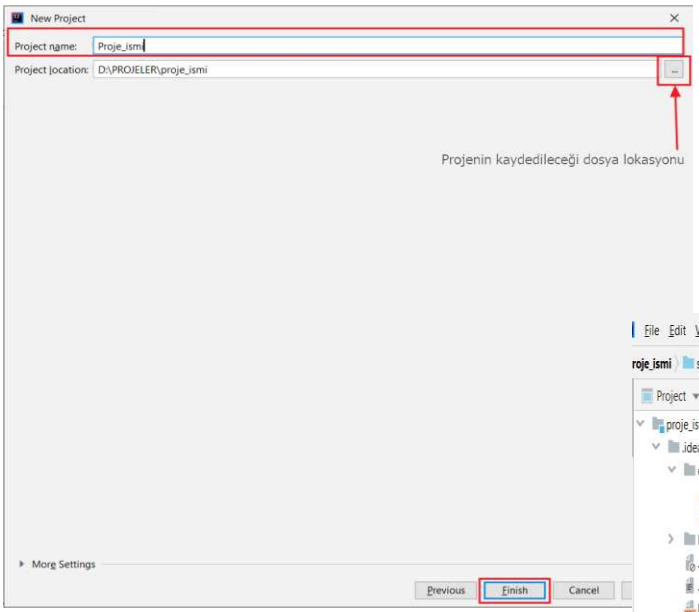
2. **IntelliJ IDEA kurulumu**
    - Kod yazmak için ıntellij idea'yı kullanacağız. Aşağıdaki linkten indirebilirsiniz.
    - https://www.jetbrains.com/idea/
    - Aşağıdaki adımları takip ederek kurulumu sağlayabilirsiniz.

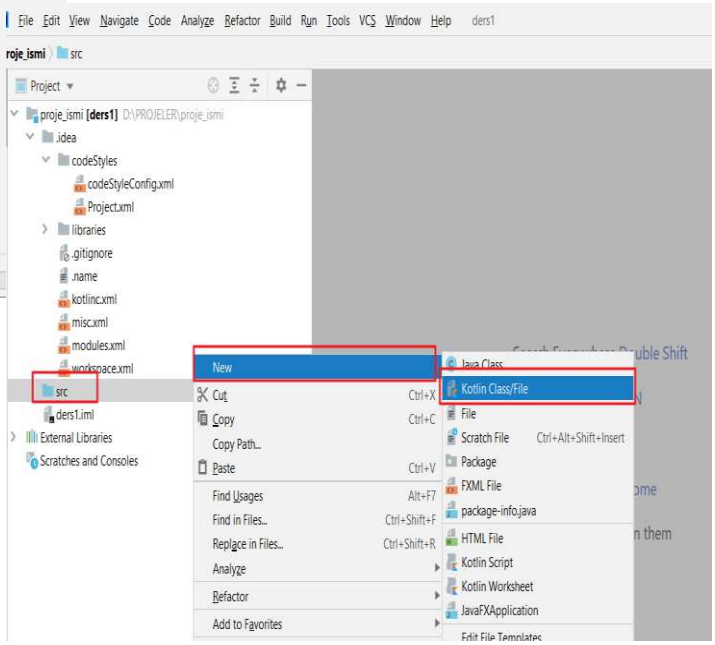


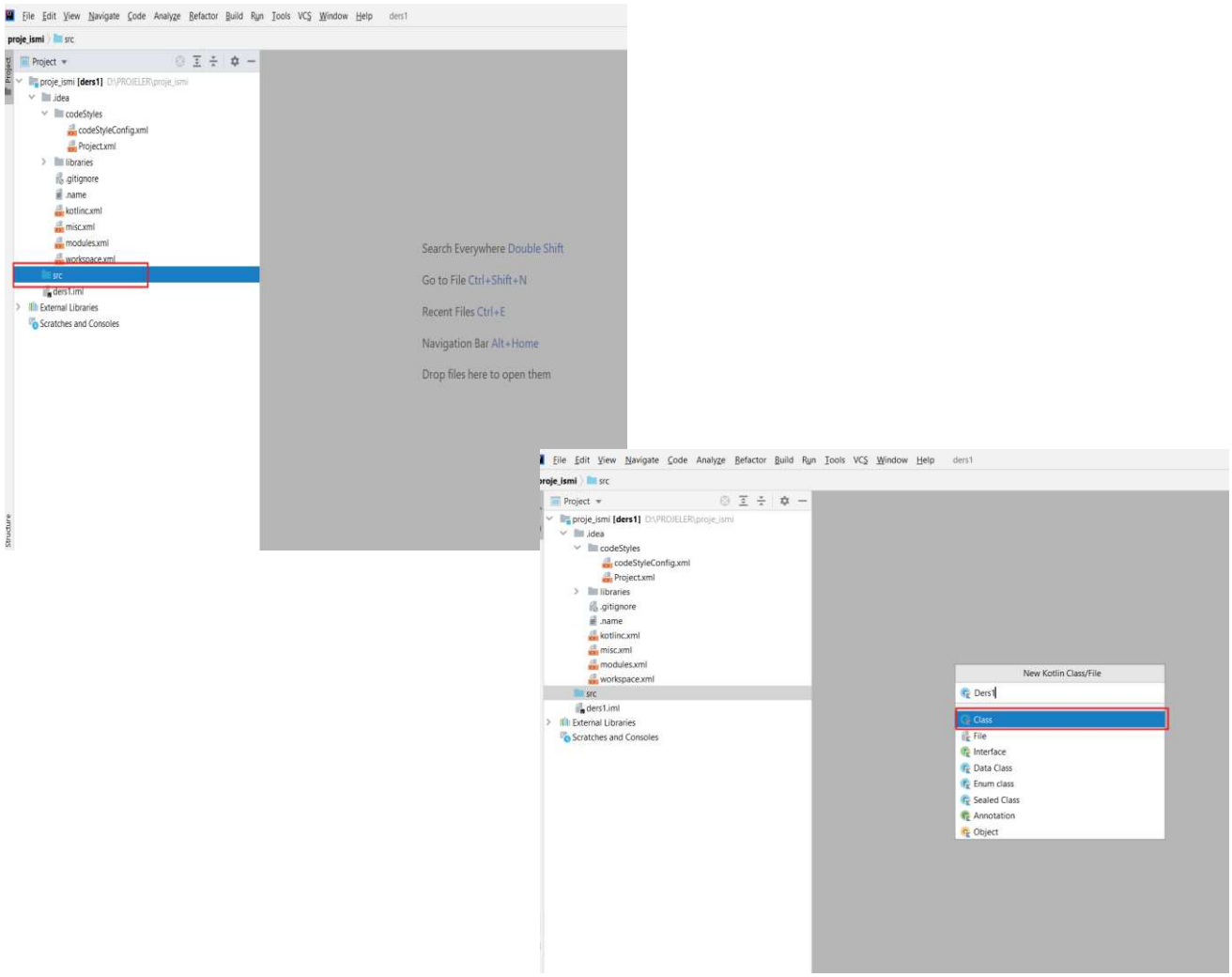
Intelli IDEA Proje Oluşturma



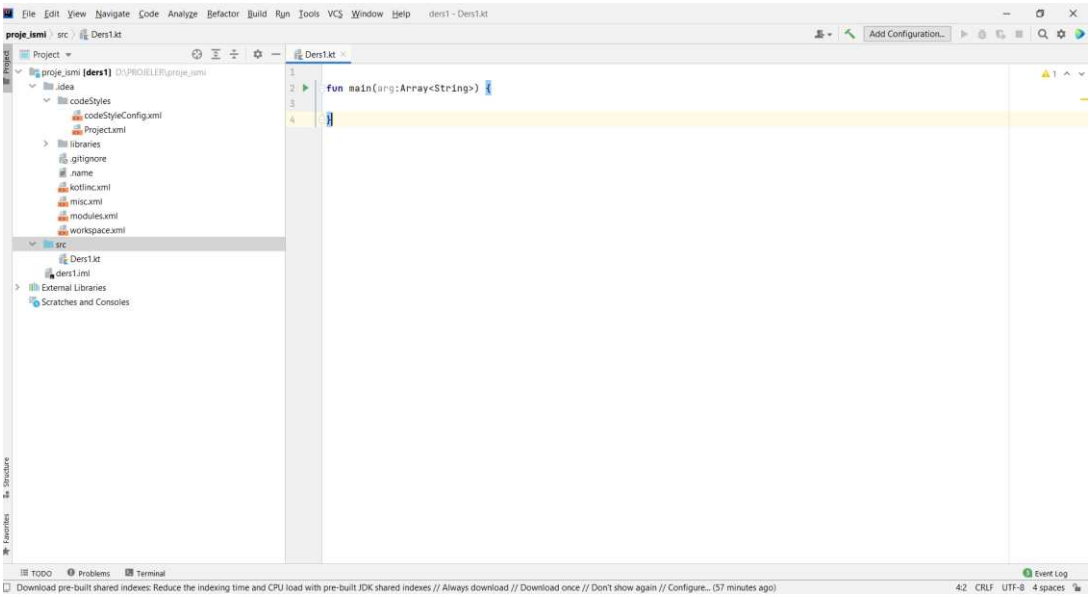


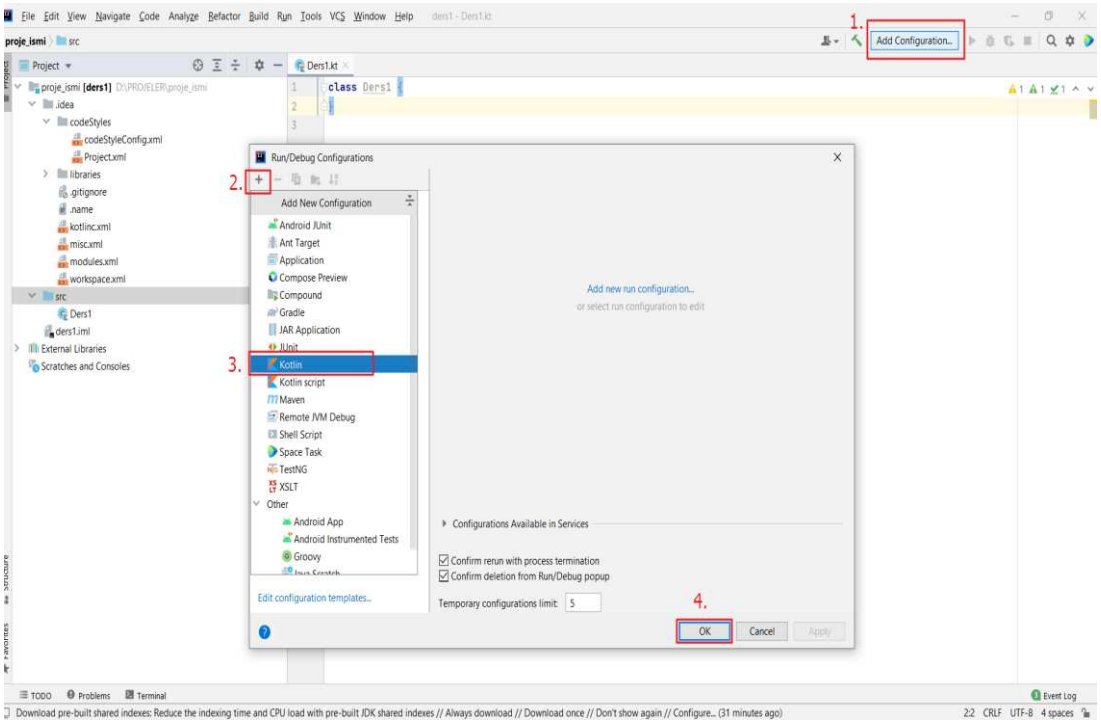


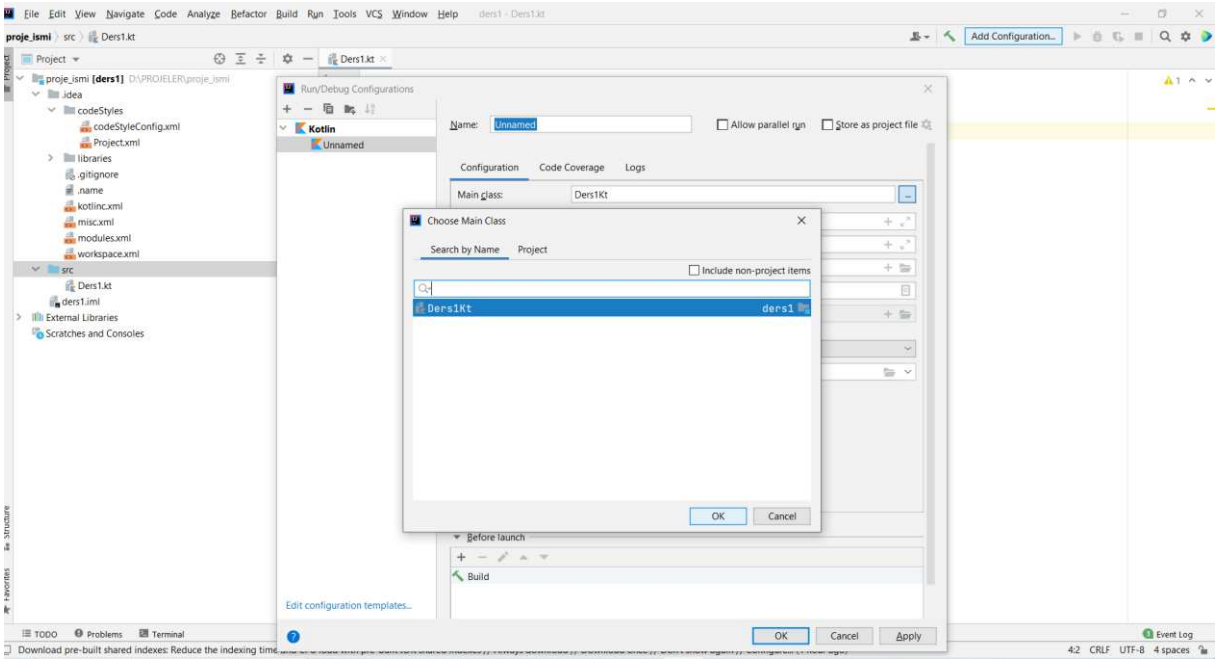




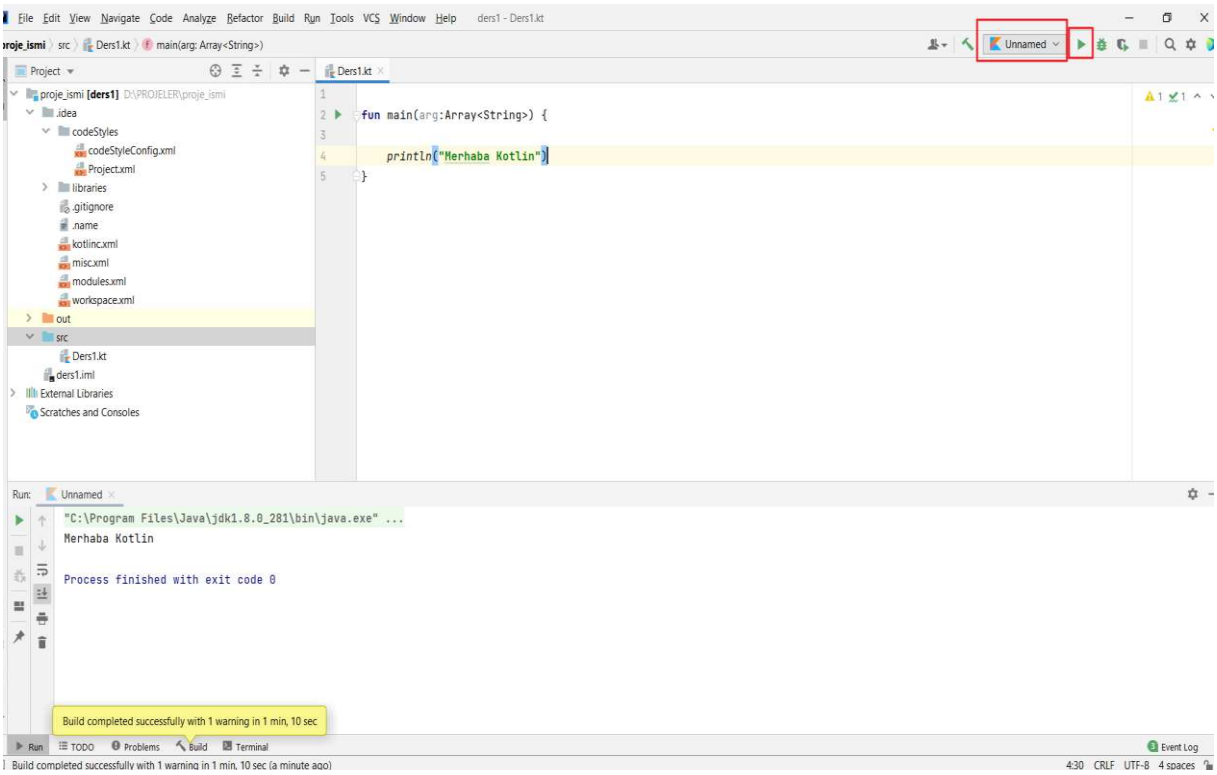
- Açılan ekranda **`fun`** `main(args :Array<String>){}` yazıp, aşağıdaki işlemleri yapmaya devam edelim.







Bu işlemleri yaptıktan sonra artık Kotlin' de kod yazabiliriz 😊



Kotlin Programını İnceleyelim

1. fun anahtar sözcüğü Kotlin'de fonksiyon oluşturmak için kullanılır. Main ana fonksiyonu, String Array türündeki argümanları kabul eden maindir. Bu , **public static void main(String args[]) işlevine** benzer.

2. Kotlin'de, argümanların veya parametrenin türü adından sonra yazılır, yani kod parçasında **arg: Array<String>.** Bu, argümanların String Array türünde olduğunu gösterir.

3. Ana fonksiyonda (fun main) herhangi bir return işlemi gerçekleşmez. Dönüş türü geçersizidir. Dönüş türümüzü tanımlamak için şu şekilde yazmalıyız.

```
fun fonksiyon adi( parametre adi : tür) : Çıktı Türü // fonksiyonun dönüş tipi
{
   return çıktı
}
```

Örnek:

```
fun yaz( mesaj: String) : String {

return "Merhaba"+ mesaj
}
```

Örnek:

```
fun main( arg: Array<String>) : String {

 println("Merhaba Kotlin")

}
```

4. println, Java'daki System.out.println() işlevine benzerdir. Ekrana yazdırma işlemini yapar.

5. Komut satırını bitirmek için noktalı virgül kullanmaya gerek yoktur.

Kotlin'de Kullanıcıdan Değer Almak

- **readLine()** komutu ile kullanıcıdan bir değer alabiliriz. Alınan değerler string tipinde veri oluyor. Sayıları kullanmak istediğimizde **toInt()** komutu ile Int tipine dönüştürebiliriz.

Örnek:

```
fun main(args :Array<String>)
{

    var adi=readLine()
    println("Adı : " +adi)

    var yasi=readLine()!!.toInt()
    println("Yaşı : " +yasi)

}
```

```
Beyza
Adı : Beyza
57
Yaşı : 57
```

Kotlin'de Tür Dönüşümü

Type conversion için numeric fonksiyonların listesi aşağıdadır.

Bu fonksiyonları kullanarak Explicit Type dönüşümü yapılır.

- **toByte()**
- **toShort()**
- **toInt()**
- **toLong()**
- **toFloat()**
- **toDouble()**
- **toChar()**

Örnek:

```
fun main(args :Array<String>)
{
    val deger:Int=15
    val longDeger:Long=deger.toLong()
    println("Long : "+longDeger)

    val doubleDeger:Double=deger.toDouble()
    println("Double Değer : "+doubleDeger)


    val doubleDeger2:Double=19.3

    val intDeger:Int=doubleDeger2.toInt()

    println("Int Değer :"+intDeger)
}
```

```
Long : 15
Double Değer : 15.0
Int Değer :19
```

Kotlin'de Range Kavramı

Range Türkçe karşılığını aralık olarak düşünebiliriz. Başlangıç değeri ve bitiş değeri olan sayı veya karakter aralıkları oluşturmayı sağlayan kavramlara Range denir.

- Range tanımlamak için **iki nokta(..)** ifadesi kullanılır.
- Aralık oluşturduktan sonra **in** anahtar sözcüğünü kullanarak her hangi bir karakter veya sayının aralıkta mı yoksa içinde değil mi onu öğrenmiş oluruz.

Örnek:

```
fun main(args :Array<String>)
{
    //5 ile 10 arasındaki sayıları ifade
    eder
    var fiveToTen=5..10

    //3 rakamı 5 ile 10 arasında mı ?
    var is3in=3 in fiveToTen

    //5 rakamı 5 ile 10 arasında mı?
    var is5in=5 in fiveToTen

    //11 sayısı 5 ile 10 arasında mı?
    var is11in=11 in fiveToTen

    println("3 değer içinde mi : $is3in")
    println("5 değer içinde mi : $is5in")
    println("11 değer içinde mi : $is11in")
```

```
3 değer içinde mi : false
5 değer içinde mi : true
11 değer içinde mi : false

Process finished with exit code 0
```

- Sadece sayılar için değil, harfler içinde Range kullanabiliyoruz.

Örnek:

```
fun main(args :Array<String>)

{
    //a ile z arasında rakamları ifade eder.
   var aToZ = 'a'..'z'

    //M rakamı a ile z arasında mı?
    var isMin = 'M' in aToZ

    //m rakamı a ile z arasında mı?
    var ismin = 'm' in aToZ


    println("M değer içinde mi : $isMin")

    println("m değer içinde mi : $ismin")
}
```

```
M değer içinde mi : false
m değer içinde mi : true

Process finished with exit code 0
```

Bu örnekte dikkat etmemiz gereken ‘a’ . . ’z’ a’dan z’ye küçük harfleri kapsar. Bütün küçük harfleri içinde barındırır. ‘m’ karakteri a ve z içerisinde yer alır. Fakat ‘M’ (büyük M) a ve z içerisinde değildir.

(`'a'..'z'` İçerisinde İngiliz alfabesi karakterlerini barındırır.)

- Artış miktarını değiştirmek için **step()** metodu kullanılır.

Örnek:

```
fun main(args :Array<String>)
{

    var oneTen=1.rangeTo(10)
    //1,2,3,4,5,6,7,8,9,10

    var newRange=oneTen.step(2)
   //artış miktarı 2 olarak değiştirildi.--> 1,3,5,7,9 oldu.

    var isTrue=6 in newRange
    //6 içerisinde mi?

    println(isTrue)
    //Sonuç ekrana yazdır

}
```

```
false

Process finished
```

Mutable ve Immutable Kavramları

- **Immutable(değişmez),** nesneler bir kez oluşturulduktan sonra içeriği değiştirilmeyen sınıflardır.
- Tam tersi olarak, değiştirilebilen sınıflar **Mutable(değiştirilebilir)** sınıflardır.
- Kısacası Immutable nesneler değiştirilemeyen nesnelerdir. Onları oluşturduktan sonra değiştiremeyiz. Bunun yerine, değişmez bir nesneyi değiştirmek istersek, onu klonlamamız ve oluştururken klonu değiştirmemiz gerekir.

Immutable nesneler, çok iş parçacıklı(multi-threaded) ortamlarda ve streamlerde kullanışlıdır. Değişmeyen nesnelere güvenmek harikadır. Başka bir thread'in nesnesini değiştiren bir iş parçacığının neden olduğu hatalar olabilir. Immutable nesneler, bu sorunların tümünü çözmüş olacaktır

Diziler (Arrays)

Birden çok veriyi bir arada saklayan veri yapılarına dizi denir. Kotlin'de dizilerin güzelliği**, karma değer** tutabilirler. Örneğin farklı tiplere sahip değişkenler için ayrı ayrı tanımlama yapılmasına gerek yoktur.

Kotlin Dilinde String

Kotlin'de String bir karakter dizisi olarak kabul edilir.

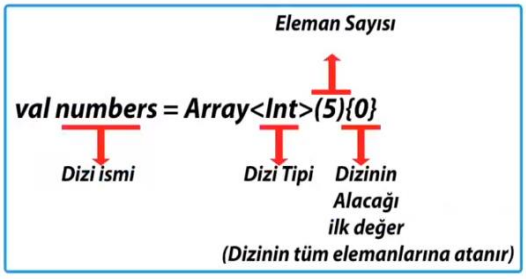
Örnek:

```
fun main()

{
    var isim="Medipol"

    for(karakter in isim)

    {
        println(karakter)

    }
}
```

```
M
e
d
i
p
o
l

Process finished with exit code 0
```

Array

Kotlin'de dizi oluşturmak için kullanmamız gereken temel sınıflardan bir tanesi Array sınıfıdır. Bu sınıf hem Java hem de Kotlin de dizi oluşturmak için kullanılır.



Örnek:

```
fun main()

{

  /*Array ile dizi tanımlamak için aşağıdaki yapıyı
    kullanmalıyız. Dizinin eleman sayısı 5 olacaktır.
    Ayrıca dizinin ilk değeri sıfırdır. Diziye sonradan
    yeni elemanlar eklenebilir.
     */


    val numbers =Array<Int>(5){0}

    /*Dizinin ilk halini ekrana yazdırmak için for döngüsü
    kullanılır. numbers.size dizinin eleman sayısını verir.
    Burada index numaraları ile çalıştığımız için i
    değişkenine sıfır ile 4 arasındaki değerleri vermeliyiz.*/


    for (i in 0..numbers.size-1){


 /*Dizi elemanları ekrana tek tek yazılır.*/

        println("numbers[$i]:"+numbers[i])

  }
}
```

```
numbers[0]:0
numbers[1]:0
numbers[2]:0
numbers[3]:0
numbers[4]:0

Process finished with exit code 0
```

Örnek:

```
fun main()
{

    val numbers=Array<Int>(5){0}

 /* Diyelim ki dizinin 3.elemanını güncellemek istiyoruz.
 Güncelleme yapmak için aşağıdaki kodu yazmalıyız. numbers[2] ile
3.elemana 25 değeri atanır. Bu işlemden sonra sıfır silinir yerine 25
değeri eklenir. */


    numbers[2]=25

    /*Dizinin içeriğini yazdırıyoruz.*/

    println("Dizinin Elemanları")


    for (i in 0..numbers.size-1){

        println("numbers[$i]:"+ numbers[i])
    }

}
```

```
Dizinin Elemanları
numbers[0]:0
numbers[1]:0
numbers[2]:25
numbers[3]:0
numbers[4]:0

Process finished with exit code 0
```

ArrayList

- Array ile ArrayList dizi oluşturmak için kullanılır. Bununla beraber ArrayList sınıfının bazı avantajları var.
- ArrayList ile oluşturulan dizinin eleman sayısını önceden belirtmek zorunda değiliz.
- Dizinin boyutu dinamik olarak belirlenir. Bu yönüyle ArrayList çok avantajlıdır.
- Bir diğer fark da, diziye eleman eklemek için **add(),** güncelleme yapmak için **set(),** dizi elemanlarına erişmek için **get()** metotlarını da kullanabilir.

Örnek:

```
fun main()
{
    /*ArrayList ile çalışırken dizi tipi mutlaka belirtilmelidir.*/

    var name= ArrayList<String>()


    /*
    ArrayList dizisine eleman eklemek için add()metodu kullanılır.
     */

    name.add("Buse")
    name.add("Baran")
    name.add("Çağatay")

    /*
    ArrayList ile oluşturulan dizideki elemanlara erişmek için
    get() metodu kullanılır. Aşağıda dizinin ilk elemanına (sıfır
    index numarasına sahip) erişmek için get() metodu index numarası
    ile kullanılmıştır.
     */

    println("name[0] :" +name.get(0))
    println("name[1] :" +name.get(1))
    println("name[2] :" +name.get(2))

    name.set(2,"Derya")

    println("name[2] :" +name.get(2))
}
```

```
name[0] :Buse
name[1] :Baran
name[2] :Çağatay
name[2] :Derya

Process finished with exit code 0
```

arrayOf()

Bu yapı esasen koleksiyonlara benzer ancak kotlin.colletions alan adı altında verilmediği için resmi olarak bir koleksiyon değildir.

Ancak normal dizi tanımlarından uzaklaşarak farklı veri tipleri ile çalışmayı desteklemektedir.

Bu özelliği ile koleksiyonlara benzerdir.

Örnek:

```
fun main()
{
    /*arrayOf() ile pratik bir şekilde karma verilerden
    oluşan diziler oluşturabiliriz
     */

    var mix =arrayOf("Kotlin",17,'b',20.5,"Java")

    println("mix[3]:    "+mix[3])
}
```

```
mix[3]:   20.5

Process finished with exit code 0
```

Örnek:

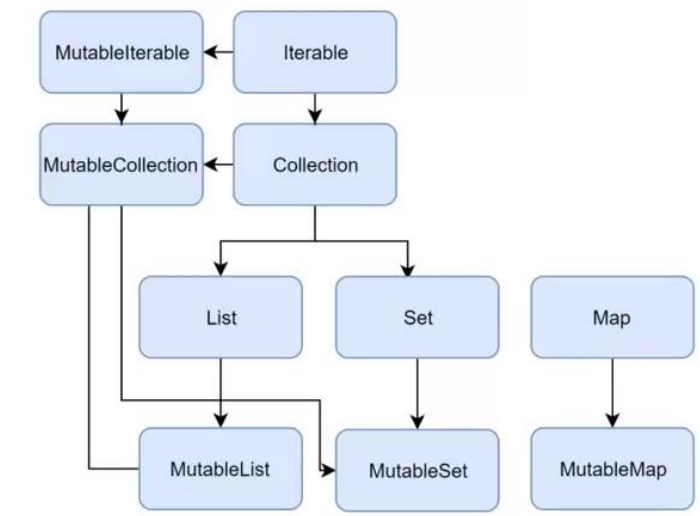
```
fun main()
{
    /*arrayOf() ile pratik bir şekilde karma verilerden
    oluşan diziler oluşturabiliriz
     */

    var mix =arrayOf("Kotlin",17,'b',20.5,"Java")

    println("mix[3]:    "+mix[3])

    /*dizinin 3.elemanını güncellemeden ekrana yazdıralım
    */
    println("mix[2] :    "+mix[2])

    //dizinin 3.elemanını 25 olarak güncelleyelim.
    //
    mix[2]=25

    println("mix[2] :    "+mix[2])

}
```

```
mix[3]:   20.5
mix[2] :   b
mix[2] :   25

Process finished with exit code 0
```

- arrayOf() karma veriler ile çalışmayı desteklediği gibi tek tip veriler ile çalışmayı da destekler. Bu yönüyle dizilere benzeyen bu yapının aşağıdaki çeşitlerini kullanarak tek tip veriler ile çalışabiliriz.

Metot	Açıklama
charArrayOf()	Sadece **Char** tipinde verilerin olduğu diziler oluşturmak için
doubleArrayOf()	Sadece **Double** tipinde verilerin olduğu diziler oluşturmak için
floatArrayOf()	Sadece **Float** tipinde verilerin olduğu diziler oluşturmak için
intArrayOf()	Sadece **Int** tipinde verilerin olduğu diziler oluşturmak için kullanılır.

Koleksiyonlar



- Her bir Collection bir değişkendir.
- Bir Collection her ne kadar mutable olursa olsun var yerine val olarak tanımlanırsa o Collection' a tekrardan atama yapılamaz!!!

List Sınıfı

Elemanlarına index leriyle (liste içerisinde bulunduğu pozisyon) ulaşabileceğimiz sıralı Collection lardır. Bir element listenin içerisinde **birden fazla kez bulunabilir.**

listOf()

Üzerinde ekleme, silme ve güncelleme işlemlerinin **yapılamadığı read-only yapılarıdır**. Bu liste değiştirilemez.

Örnek:

```
fun main()
{

    val isimler = listOf("Beyza", "KOYULMUS", "Rabia","Rabia")

    for(item in isimler)
    {
        println(item)
    }
}
```

```
Beyza
KOYULMUS
Rabia
Rabia

Process finished with exit code 0
```

- listOf(), null değeri alabilir.

Örnek:

```
fun main()
{

    val isimler = listOf("Beyza", "KOYULMUS",
"Rabia","Rabia")

    println("İsimler boyutu:" + isimler.size )


    val isimler2 =listOf("Beyza", null,null,
"Rabia")

    println("İsimler2 boyutu:"+ isimler2.size)
}
```

```
İsimler boyutu:4
İsimler2 boyutu:4

Process finished with exit code
```

- listOf ile oluşturulan listelerde null değerleri bir eleman gibi görülür. Bu yüzden boyut sorulduğunda null değerlerinide ekler.

- Liste içerisinde null değerinin bir eleman gibi görülmesi **istenmediği durumda** listOfNotNull() kullanılır.

listOfNotNull()

listOfNotNull(), eleman sayısına null olan değerleri katmaz.

Örnek:

```
fun main()
{

    val isimler3 = listOfNotNull("Beyza", "KOYULMUS",null,null,"Rabia")

    println("İsimler3 boyutu:" + isimler3.size )

}
```

```
İsimler3 boyutu:3

Process finished with exit code 0
```

- Örnekte görüldüğü gibi null değerleri eleman olarak sayılmamıştır. Aynı örnekte ikinci elemanın hangi eleman olduğunu bulalım. (İndislerin sıfır sayısı ile başladığını unutmayalım!)

Örnek:

```
fun main()
{

    val isimler3 = listOfNotNull("Beyza", "KOYULMUS",null,null,"Rabia")

    println("İsimler3 boyutu:" + isimler3.size )

    println("İsim :"+isimler3.get(2))

}

//isimler3.get(2) ikinci elemanı getirir.
```

```
İsimler3 boyutu:3
İsim :Rabia

Process finished with exit code 0
```

arrayListOf()

- Dinamik bir dizi yapısıdır.
- Sonradan eklemeler, baştan eklemeler gibi eklemeler yapılabilir.
- listOf metodunda herhangi bir ekleme çıkarma yapamıyorduk.
- arrayListOf() metodu ile ekleme çıkarma vs. yapabiliriz.

Örnek:

```
fun main()
{

    val isimler =arrayListOf<String>("Beyza",
"Elif","Yiğit","Görkem","Rabia")

    isimler.add("Ali")

    //isimler listesine Ali ekliyor.

    for(item in isimler)
    {
      println(item)
    }
}
```

```
Beyza
Elif
Yiğit
Görkem
Rabia
Ali

Process fin
```

.add() metodu ile listenin sonuna ekleme yapabildiğimiz gibi, istediğimiz indise de ekleme yapabiliriz. Örneğin Ali ismini 1.indis olarak yazalım.(2.eleman oluyor)

Örnek:

```
fun main()
{

    val isimler =

arrayListOf<String>("Beyza","Elif","Yiğit","Görkem","Rabia")

    isimler.add(1,"Ali")

    for(item in isimler)
    {

        println(item)

    }


}
```

```
Beyza
Ali
Elif
Yiğit
Görkem
Rabia

Process fin
```

mutableListOf()

- Üzerinde ekleme, silme ve güncelleme yapılabilen listelerdir.

Güncelleme Örnek:

```
fun main()
{
    val isimler = mutableListOf("Beyza", "Elif","Yiğit","Görkem","Rabia")

    //Güncelleme
    isimler[0]="beyza2"

    for(item in isimler)
    {
        println(item)
    }
}
```

```
beyza2
Elif
Yiğit
Görkem
Rabia

Process finished
```

Ekleme Örnek:

```
fun main()
{

    val isimler = mutableListOf("Beyza", "Elif",   "Yiğit", "Görkem",
"Rabia")

    //Ekleme
    isimler.add("beyza2")


    for(item in isimler)
    {
        println(item)
    }

}
```

```
Beyza
Elif
Yiğit
Görkem
Rabia
beyza2

Process finished
```

Silme Örnek:

```
fun main()
{

  val isimler = mutableListOf("Beyza", "Elif",  "Yiğit","Görkem","Rabia")

    //Silme
    isimler.removeAt(1)

    for(item in isimler)
    {
        println(item)
    }
}
```

```
Beyza
Yiğit
Görkem
Rabia

Process finished
```

Karıştırma Örnek:

```
fun main()
{

    val isimler = mutableListOf("Beyza", "Elif", "Yiğit","Görkem","Rabia")

    //Listeyi Karıştır
    isimler.shuffle()

    for(item in isimler)
    {
        println(item)
    }
}
```

```
Görkem
Beyza
Elif
Rabia
Yiğit

Process finished
```

Set Sınıfı

- Elemanları birbirinden benzersiz olan Collectionlardır.
- Yani **bir eleman bir set içerisinde iki kez bulunamaz.**
- Sıralı olup olmaması önemsizdir.
- Alfabe örnek olarak gösterilebilir.

Set sınıfını, matematikteki kümeler konusu olarak düşünebiliriz. Örneğin iller kümesi için her bir ilden yalnızca bir adet bulunabilir.

setOf()

- Ekleme, çıkarma ve güncelleme **yapılamayan** yapılardır.
- Elementler eşsiz olduğundan dolayı her bir set sadece bir tane null'a sahip olabilir.
- Eğer iki setin boyutları ve elemanları birbirine eşit ise o zaman bu iki set birbirine eşit olur.

Örnek: Elemanları aynı fakat sıraları farklı olan iki kümenin eşitliklerini inceleyelim.

```
fun main()
{

    val isimler1 = setOf("Beyza","Elif","Yiğit")

    val isimler2 = setOf("Elif","Yiğit","Beyza")

    println(isimler1==isimler2)
}
```

```
true

Process finished
```

mutablesetOf()

- Üzerinde ekleme, çıkarma ve güncelleme yapılabilen setlerdir.

  ```
  var isimler1 = mutableSetOf("Beyza","Elif","Yiğit")
  ```

  şeklinde tanımlanır.

Map Sınıfı

- key – value (anahtar – değer) şeklinde verilerden oluşan collectionlardır.
- Her key benzersizdir ve ancak bir değeri gösterebilir. Ancak value lerin eşsiz olmasına gerek yoktur.

mapOf()

- Üzerinde ekleme, güncelleme, silme işlemi **yapılamayan** yapılardır.
- Collection arayüzünü inherit etmemesine rağmen bir collection type olarak sınıflandırılır.
- Key – value şeklinde veri tutar.
- Key üzerinden value ye ulaşılabilir.

Örnek: Bazı il ve plakaları mapOf ile tanımlayalım.

```
fun main()
{

    var sehirlerMap = mapOf("İstanbul" to 34,"Bursa" to 16,"Eskişehir" to
26,"Antalya" to 7)

    //Ekrana bütün keyler yazdırılır- İstanbul, Bursa, Eskişehir, Antalya
    println("All keys : ${sehirlerMap.keys}")


    //Ekrana bütün value ler yazdırılır- 34, 16, 26, 7
    println("All value : ${sehirlerMap.values}")


}
```

```
All keys : [İstanbul, Bursa, Eskişehir, Antalya]
All value : [34, 16, 26, 7]

Process finished with exit code 0
```

hashMapOf()

- Üzerinde ekleme, güncelleme, silme işlemlerinin yapıldığı maplerdir.

Örnek:

```
fun main()
{

    var kisiHashMap = hashMapOf("Adı" to "Beyza","Yasi" to 59,"Sehir" to
"İstanbul")

    //Ekrana bütün keyler yazdırılır- Adı, Yasi, Sehir
    println("All keys : ${kisiHashMap.keys}")

    //Ekrana bütün value ler yazdırılır- Beyza, 59, İstanbul
    println("All value : ${kisiHashMap.values}")

}
```

```
All keys : [Sehir, Adı, Yasi]
All value : [İstanbul, Beyza, 59]

Process finished with exit code 0
```

mutableMapOf()

- Üzerinde ekleme, güncelleme, silme işlemlerinin yapıldığı maplerdir.

Örnek:

```
var kisiMap = mutableMapOf("Beyza" to 81,"Hamza" to 43,"Ayşe" to 42)
    println(kisiMap)

    //Ekleme- Bu yazım şeklinin JetBrains tarafından indexli yazmaya
çevrilmesi önerilmektedir.
    kisiMap.put("Ali",17)

    //Indexli Ekleme
    kisiMap["Beyza"]=26

    //Silme
    kisiMap.remove("Ayşe")

    //Güncelleme
    kisiMap["Hamza"]=16

    println(kisiMap)
}
```

```
{Beyza=81, Hamza=43, Ayşe=42}
{Beyza=26, Hamza=16, Ali=17}

Process finished with exit code 0
```

Collection Types(Tipleri) Özet

List :	Elemanlarına indexleriyle (liste içerisinde bulunduğu pozisyon) ulaşabileceğimiz sıralı collectionlardır. Bir element listenin içerisinde birden fazla kez bulunabilir.
Set :	Elemanları birbirinden benzersiz olan collectionlardır. Yani bir eleman bir set içerisinde iki kez bulunamaz. Sıralı olup olmaması önemsizdir. Alfabe örnek olarak gösterilebilir.
Map :	Key – value (anahtar-değer) şeklinde verilerden oluşan collectionlardır. Her key benzersizdir ve bir değeri gösterebilir. Ancak valuelerin eşsiz olmasına gerek yoktur.

En Çok Kullanılan Array Fonksiyonları:

- **add()**
- **get()**
- **remove()**
- **removeAt()**
- **contains()**
- **isEmpty()**
- **clear()**
- **indexOf()**
- **equals()**
- **size**

Bu fonksiyonlar, Array fonksiyonlarının bazılarıdır. Genellikle bu fonksiyonlar kullanılır.

Array : add()

ArrayList'e item eklemek için **add()** metodunu kullanırız.

Örnek:

```
fun main()
{
 val arrayList = ArrayList<String>()
     arrayList.add("Beyza")
     arrayList.add("Kotlin")
     arrayList.add("Kotlin Dersleri")

     println(arrayList)
     arrayList.add(1,"yeni item 1")
     println(arrayList)
}
```

```
[Beyza, Kotlin, Kotlin Dersleri]
[Beyza, yeni item 1, Kotlin, Kotlin Dersleri]

Process finished with exit code 0
```

Örnek:

```
fun main()
{

 val arrayList2 = ArrayList<Int>()

   arrayList2.add(2)
   arrayList2.add(5)
   arrayList2.add(5)
   arrayList2.add(5)

   println(arrayList2)

   arrayList2.add(2,7)
   println(arrayList2)
}
```

```
[2, 5, 5, 5]
[2, 5, 7, 5, 5]

Process finished with exit code 0
```

Array : get()

ArrayList'teki bir itemi almak için get() metodu yada [] köşeli parantez içinde değerini almak istediğimiz indexi yazarız.

Örnek:

```
fun main() {

  val arrayList =ArrayList<String>()

  arrayList.add("Beyza")
  arrayList.add("Ayşe")
  arrayList.add("Ali")


  var deger1:String =arrayList.get(0) //ArrayListin 0.indisteki elemanını
almak için
  var deger2:String =arrayList.get(1) //ArrayListin 2.elemanını almak
için.
  var deger3:String =arrayList[2] //ArrayListin 3.elemanını almak için

  //Elemanların sıfırıncı indisten başladığını unutmayalım!!


  println(deger1) //Beyza
  println(deger2) //Ayşe
  println(deger3) //Ali

}
```

```
Beyza
Ayşe
Ali

Process finished with exit code 0
```

Array : remove() removeAt()

ArrayLis'ten item silmek için remove() ve removeAt(), methodlarını kullanırız.

Örnek:

```
fun main() {

  val arrayList =ArrayList<String>()

  arrayList.add("Kotlin")

  arrayList.add("Beyza")

  arrayList.add("Kotlin Dersleri")

  arrayList.add("Ayşe")


  println(arrayList)


  arrayList.remove("Beyza")

  println(arrayList)

  arrayList.removeAt(0)

  println(arrayList)


}
```

```
[Kotlin, Beyza, Kotlin Dersleri, Ayşe]
[Kotlin, Kotlin Dersleri, Ayşe]
[Kotlin Dersleri, Ayşe]

Process finished with exit code 0
```

Array : size() isEmpty() clear () contains() indexOf()

size():	ArrayList' in uzunluğunu dönen bir değişkendir. Yani kaç item eklenmiş olduğu bilgisini verir.
isEmpty():	ArrayList'in içinde item var mı yok mu yani arrayList boş mu değil mi bilgisini döner.
clear():	ArrayList'in tüm elemanları siler.
contains():	Bir itemin arrayList'in içinde olup olmadığı bilgisini boolean olarak geri döner. Eğer item arrayList'te mevcutsa **true**, değilse **false** döner.
indexOf():	ArrayList'imizin içinde aradığımız itemin indexsini verir.Eğer aradığımız item arrayListte yoksa -1 değeri döner.

Örnek:

```
fun main() {

  val arrayList =ArrayList<String>()

  arrayList.add("Kotlin")
  arrayList.add("Beyza")
  arrayList.add("Kotlin Dersleri")
  arrayList.add("Ayşe")


  println("Eleman sayısı : " + arrayList.size)
  println("Liste Boş mu? " +arrayList.isEmpty())

  arrayList.clear()

  println("Liste temizlendi")

  println("Liste boş mu? : " +arrayList.isEmpty())


}
```

```
Eleman sayısı : 4
Liste Boş mu? false
Liste temizlendi
Liste boş mu? : true

Process finished with exit code 0
```

Örnek: contains() ve indexOf() metodu kullanımı

```
fun main() {

  val arrayList =ArrayList<String>()

  arrayList.add("Kotlin")
  arrayList.add("Beyza")
  arrayList.add("Kotlin Dersleri")
  arrayList.add("Ayşe")



  println(arrayList.contains("Beyza"))
  //Beyza, liste içerisinde mevcut olduğu için true


  println(arrayList.contains("Merve"))
  //Merve, liste içerisinde mevcut olmadığı için false


  println(arrayList.indexOf("Beyza"))
  //Beyza kaçıncı eleman ? 1


  println(arrayList.indexOf("Ayşe"))
  //Ayşe kaçıncı eleman? 3


  println(arrayList.indexOf("Ali"))
 //Ali listede olmadığı için -1



}
```

```
true
false
1
3
-1

Process finished with exit code 0
```

Array : equals()

İki ArrayList'i karşılaştırmak için kullanılır, yani ArrayListler aynı elemanlara sahip mi, değil mi?

Örnek:

```
fun main() {

  val arrayList1 =ArrayList<String>()
  arrayList1.add("Kotlin")
  arrayList1.add("Beyza")

    val arrayList2 =ArrayList<String>()
  arrayList2.add("Kotlin")
  arrayList2.add("Beyza")

    println(arrayList1.equals(arrayList2))  //true
    //arraylist1 ve arrayList2 eşit mi

}
```

```
true


Process finished with exit code 0
```

- Eleman sıralarıda önemlidir. Aynı elemanlar var fakat farklı indislerde olsaydı false olurdu

Örnek:

```
fun main() {

  val arrayList1 =ArrayList<String>()

  arrayList1.add("Kotlin")
  arrayList1.add("Beyza")


  val arrayList2 =ArrayList<String>()

  arrayList2.add("Beyza")

  arrayList2.add("Kotlin")

  println(arrayList1.equals(arrayList2))  //false
 //arraylist1 ve arrayList2 eşit mi

}
```

```
false


Process finished with exit code 0
```

Koleksiyon Fonksiyonları

- any
- count
- max
- min
- maxBy
- minBy
- filter
- contains
- elementAt
- first
- indexOf
- last
- indexOfLast
- indexOfFirst
- reverse
- sort
- sortDescending
- groupBy

any(): Listedeki elemanlardan herhangi biriyle verilen koşul sağlanıyorsa geriye true döndürür.

Örnek:

```
fun main() {

  val list = listOf(1, 2, 3, 4, 5, 6)
  println(list.any { it % 2 == 0}) //Return True
  //listedeki en az bir eleman mod 2 de 0 sonucunu verir
  //örneğin listedeki 4, bu koşulu sağlar.Bu yüzden True döndürür.

  println(list.any { it > 10 }) //Return False
  //Lşstedeki hiçbir eleman 10 dan büyük değildir.

  println(list.any {it < 3}) //Return True
  //Listedeki 1 ve 2 bu koşulu sağlar.

}
```

```
true
false
true

Process finished with exit code 0
```

count() : Verilen koşula uyan listedeki eleman sayısını verir.

Örnek:

```
fun main() {
  val list = listOf(1, 2, 3, 4, 5, 6)

  println(list.count()) //Return 6

  println(list.count{ it % 2 == 0 }) // Return 3

  println(list.count{ it > 10 }) //Return 0

}
```

```
6
3
0

Process finished with exit code 0
```

max(): Listedeki en büyük elemanı döndürür. Eğer listede eleman yoksa geriye null değeri döndürür.

maxBy(): Verilen koşulun en büyük ilk değerini döndürür. Yoksa geriye null değeri döndürür.

Örnek:

```
fun main() {

  val list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
  println(list.max())
  //Listedeki en büyük sayı

  println(list.maxBy{it%2})
  // mod 2 de, büyük değer alan ilk sayı

  println(list.maxBy { it > 3 })
  //3 ten büyük listedeki ilk büyük sayı 4'tür

  println(list.maxBy { it < 3 })
  //3 ten küçük listedeki ilk sayı 1'dir

}
```

```
10
1
4
1

Process finished with exit code 0
```

Önemliler!!!

filter() : Listedeki elemanların verilen koşul durumuna uyanları geriye döndürür.

contains() : Eğer eleman listede bulunuyorsa true değerini döndürür.

elementAt : Listede indisteki elemanı döndürür. Eğer parametrede verdiğimiz değer listenin uzunluğundan büyükse veya negatif değerse ArrayIndexOutOfBoundsException hatası verir.

Örnek:

```
fun main() {

  val list = listOf(2, 4, 3, 5, 7)
  println(list.filter { it % 2 == 0 })
 // İkiye bölünen 2 ve 4 olduğu için : Return [2,4]

  println(list.contains(2))
 // Listede 2 elemanı olduğu için : Return True

  println(list.contains(-1))
 //Listede -1 elemanı olmadığı için :Return False

  println(list.elementAt(2)) //Print 3
 //Listedeki 2.indis 3'tür. İndisler sıfırdan başlar.

  println(list.elementAt(7))
 // 7 eleman olmadığı için hata verir. Liste uzunluğu 5

}
```

```
[2, 4]
true
false
3
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException Create breakpoint : 7
```

first() : Verilen koşula uygun listedeki ilk elemanı döndürür. Eğer koşula uygun eleman listede yoksa program NoSuchElemantException hatası verir.

Örnek:

```
fun main() {

  val list = listOf(2, 4, 3, 5, 7, 2, 4, 0, 4, 6, 13)

  println(list.first{ it % 2 == 0 }) //Return 2

  println(list.first{ it % 2 == 1 }) //Return 3

  println(list.first {it / 2 == 4 }) //HATA: NoSuchElementException

}
```

```
2
3
Exception in thread "main" java.util.NoSuchElementException:
```

last() : Verilen koşula uygun listedeki son elemanı döndürür. Eğer koşula uygun eleman listede yoksa program NoSuchElemantException hatası verir.

Örnek:

```
fun main() {
  val list = listOf(2, 4, 3, 5, 7, 2, 4, 0, 4, 6, 13)
  println(list.last { it % 2 == 0 }) //Return 6
  println(list.last { it % 2 == 1 }) //Return 13
  println(list.last {it / 2 == 4 }) //HATA: NoSuchElementException
}
```

```
6
13
Exception in thread "main" java.util.NoSuchElementException
```

Index Fonksiyonları

indexOf() : Verilen değeri listenin içinde arar ve ilk bulduğu yerdeki indis bilgisini geriye döndürür. Eğer listenin içinde aranan değer yoksa geriye -1 değerini döndürür.

Örnek:

```
fun main() {
  val list = listOf(2, 4, 3, 5, 7, 2, 4, 0, 4, 6, 13)
  println(list.indexOf(7))
  println(list.indexOf(-5))
}
```

```
4
-1

Process finished with exit code 0
```

indexOfFirst() : Verilen koşula uygun listedeki ilk elemanı döndürür. Eğer koşula uygun eleman listede yoksa geriye -1 değerini döndürür.

Örnek:

```
fun main() {

  val list = listOf(2, 4, 3, 5, 7, 2, 4, 0, 4, 6, 13)

  println(list.indexOfFirst { it % 3 == 0 })

  println(list.indexOfFirst { it * 5 == 1 })

}
```

```
2
-1

Process finished with exit code 0
```

indexOfLast() : Verilen koşula uygun listedeki son elemanı döndürür. Eğer koşula uygun eleman listede yoksa geriye -1 değerini döndürür.

Örnek:

```
fun main() {

  val list = listOf(2, 4, 3, 5, 7, 2, 4, 0, 4, 6, 13)
  println(list.indexOfLast { it % 3 == 0 })

  println(list.indexOfLast { it * 5 == 1 })

}
```

```
9
-1

Process finished with exit code 0
```

Koleksiyon: Sıralama Fonksiyonları

reverse () : Listeyi terse çevirir.

sort() : Listeyi küçükten büyüğe doğru sıralar.

sortDescending(): Listeyi büyükten küçüğe doğru sıralar.

groupBy() : Verilen koşul durumuna göre listedeki elemanları bölerek geriye cap türünde bir collection döndürür.

Örnek:

```
fun main() {

  val list = mutableListOf(2, 1, 3, 5, 4)
  println(list) // [2, 1, 3, 5, 4]

  list.reverse() // listeyi tersine çeviriyor
  println(list)  // [4, 5, 3, 1,2]

  list.sort() // Liste elemanlarını küçükten büyüğe sıralar.
  println(list) // [1, 2, 3, 4, 5]

  list.sortDescending() //Liste elemanlarını büyükten küçüğe sıralar
  println(list) // [5, 4, 3, 2, 1]

  println(list.groupBy { if (it % 2 == 0)"cift" else "tek" })
  //{tek=[5, 3, 1], cift=[4, 2] }
}
```

```
[2, 1, 3, 5, 4]
[4, 5, 3, 1, 2]
[1, 2, 3, 4, 5]
[5, 4, 3, 2, 1]
{tek=[5, 3, 1], cift=[4, 2]}

Process finished with exit code 0
```

String İşlemleri : Birleştirme

String türünde değişkenleri birleştirmek istediğimizde + sembolü ile veya “ “ içerisinde değişkenlerin başına $ işareti koyarak yapabiliriz.

Örnek:

```
fun main() {

  val string1 = "Medipol "
  val string2 = "Üniversitesi "
  val sayi = 34
  val string3 = string1 + string2 + sayi

  println(string3)

  val string4 ="$string1 $string2 $sayi"

  println(string4)
}
```

```
Medipol Üniversitesi 34
Medipol  Üniversitesi  34

Process finished with exit code 0
```

String Fonksiyonları

get(int index) : String içerisinde belirtilen index değerindeki karakter değerini verir.

substring(int beginIndex): Verilen beginIndex değerindeki indexten sonrasını verir.

substring(int beginIndex, int endIndex): Verilen begin index ile end Index arasındaki bölümü verir.

equals(Object another) : Object olarak verilen değerin varolan değerle eşitliğini kontrol eder. True | False döner.

isEmpty(): String değer boş mu dolu mu kontrolü yapar. True | False döner.

plus(String str) : String değerin sonuna farklı bir string değer eklemek için kullanılır.

replace(char old, char new) : String metin içerisindeki belirtilen oldChar değerlerini newChar olarak değiştirir.

Örnek:

```
fun main() {

  val string1 = "Medipol Üniversitesi"

  //String değerden bir karakter çekme
  println(string1.get(3)) //4.karakteri verecektir -> i

  //Substring Kullanımı
  println(string1.substring(2)) // -> dipol Üniversitesi
  println(string1.substring(8,12)) // -> Üniv

  //equals kullanımı
  println(string1.equals("Medipol Üniversitesi"))// true
  println(string1.equals("Kotlin")) // false

  //isEmpty Kullanımı
  println(string1.isEmpty()) // false
  println("".isEmpty()) // true

  //plus kullanımı
  println(string1.plus(" Kotlin Dersleri")) //-> Medipol Üniversitesi
Kotlin Dersleri

  //replace kullanımı
  println(string1.replace(" Kotlin Dersleri", " ")) // ->Medipol
Üniversitesi
}
```

```
i
dipol Üniversitesi
Üniv
true
false
false
true
Medipol Üniversitesi Kotlin Dersleri
Medipol Üniversitesi
```

String İşlemleri : toLowerCase () toUpperCase()

Lower: aşağı Upper:yukarı

toLowerCase() : Verilen stringi küçük harf yapar.

toUpperCase() : Verilen stringi büyük harf yapar.

Örnek:

```
fun main() {

  val string1 = "Medipol Üniversitesi"
  println(string1.toLowerCase()) //medipol üniversitesi
  println(string1.toUpperCase()) //MEDİPOL ÜNİVERSİTESİ

}
```

```
medipol üniversitesi
MEDİPOL ÜNİVERSİTESİ


Process finished with exit code 0
```

Scope Fonksiyonlar :apply, also, run, let, with

Bir nesnenin bağlam içerisinde bir kod bloğunu çalıştırmasını sağlayan fonksiyonlardır.

Bu fonksiyonların ortak özellikleri nesne üzerinde çağrılmaları ve lambda ifadesini parametre olarak kullanmalarıdır.

Lambda -> : Referans verilebilir ve tekrar tekrar kullanılabilir ifadelerdir. Lambda fonksiyonları, diğer bir fonksiyona argüman olarak iletilebilir.

Context Nesneleri (Objects) : it veya this

Oluşturulan bağlamda nesnenin referansına olan erişim hakkında bilgi verir.

İki adet erişim yolu vardır:

- **this (lambda alısıcı)**
- **it (lambda argümanı)**

Her iki kullanımda aynı özellikler bulunmaktadır tek fark nesneye olan referansın isimlendirilme biçimidir.

let

- Nesneye erişim lambda argümanı it vasıtasıyla gerçekleşir ve geriye lambda sonucunu döndürür.
- let fonksiyonunda genelde null kontrolü yapılıp, değer null değilse belirlenmiş bir kod bloğunun çalıştırılmasında kullanılır.

Örnek:

```
fun main() {

  var isim: String? = null

  isim ="Beyza"

  isim?.let{
    println(it)
  }

}
```

```
fun main() {

  var isim: String? = null

  isim ="Beyza"

  if(isim.isNullOrEmpty())
  {
    println(isim)
  }

}
```

```
Beyza


Process finished with exit code 0
```

örnekteki `if` ve `let` kullanımları aynı işlevi görür. Let yapısı karışık görünümü engeller. Temel amacı **null kontrolü** yapmaktır. İkinci bir görevi ise **atama** yapmaktır.

Örnek: kullanicilar isimli liste içerisinden 4 karakterden büyük olan isimleri ozelKullanicilar adlı değişkene atayalım.

```
fun main() {

  val kullanicilar: List<String> =
mutableListOf("Beyza","Ceren","Buse","Ali","Mert","Ayşe")

  val ozelKullanicilar = kullanicilar.filter { it.length > 4 }.let{it}
    println(ozelKullanicilar)

}
```

```
[Beyza, Ceren]


Process finished with exit code 0
```

Örnek: kullanicilar isimli liste içerisinden 4 karakterden büyük olan isimlerden **ilkini** ozelKullanicilar adlı değişkene atayalım.

```
fun main() {

  val kullanicilar: List<String> =
mutableListOf("Beyza","Ceren","Buse","Ali","Mert","Ayşe")

  val ozelKullanicilar = kullanicilar.filter { it.length > 4 }.let{it[0]}
    println(ozelKullanicilar)

}
```

```
Beyza

Process finished with exit code 0
```

run

- Nesneye erişim this ifadesiyle sağlanır ve geriye lambda sonucunu döndürür.
- run ile let fonksiyonları birbirine benzemektedir.İki fonksiyonda geriye lambda sonucunu döndürmektedir.
- Bu iki fonksiyonu birbirinden özellik ise nesneye erişimde let ifadesinde it kullanılır, run ise this kullanmaktadır. Her iki fonksiyonu kullanarak kodu sade ve okunabilir hale getirebiliriz.

Örnek:

```
fun main() {

  var isim: String? = null
  isim ="Beyza"

  isim?.run{
    print(this)
  }

}
```

```
Beyza
Process finished with exit code 0
```

Örnek:

```
fun main() {

  val numaralar= mutableListOf<Int>(0,0)
  val numaraAdedi=numaralar.run{

      add(1)
      add(2)
      this

  }

  println(numaraAdedi)
}
```

```
[0, 0, 1, 2]

Process finished with exit code 0
```

with(...)

- Diğer fonksiyonlardan farklı olarak, işleme girecek olan nesne with fonksiyonuna bir parametre verir. İkinci parametre ise bu nesne üzerinde yapılması istenen lambda ifadesidir.
- With, yazılan kodu yalınlaştırmak ve gereksiz tekrarlardan korunmak için kullanılır.

Örnek: numaralar adlı boş bir listeye with fonksiyonunu kullanarak elemanlar ekleyelim.

```
fun main() {

  val numaralar= mutableListOf<Int>()
  fun numaraEkle() =with(numaralar){
    add(1)
    add(2)
    add(3)
    add(8)
    this
  }
  println(numaraEkle())
}
```

```
[1, 2, 3, 8]

Process finished with exit code 0
```

apply

- Nesneye erişim this ifadesiyle sağlanır ve geriye nesnenin kendisini döndürür.
- apply, üzerinde çağrıldığı bir nesne bağlamındaki kod bloğunu çalıştırır ve nesnenin kendisini geriye döndürür.
- *apply, öncelikli hedefi objenin üyeleri için değer atama işlemi yapan kod blokları için kullanılabilir.
- apply ile daha kolay ve okunabilir şekilde atama yapabiliriz.

Örnek:

```
fun main() {

  val numaralar = mutableListOf<Int>()
  val eklenmisler = numaralar.apply {
    add(1)
    add(2)
  }
  println(eklenmisler)
}
```

```
[1, 2]


Process finished with exit code 0
```

also

- Nesneye erişim için it ifadesi kullanılır ve geriye nesneyi döndürür.
- also, almış olduğu lambda ifadesini nesne içinde çalıştırır ve bu nesneyi geri döndürür.
- Nesneye it ifadesi ile erişilir.
- also, lambda parametresi üzerinde yapılacak olan işlemlerde tercih edilir.

Örnek:

```
fun main() {

val numaralar = mutableListOf<Int>(2,3,4,5,6)
  numaralar.also{
    println(it.get(3))
  }
}
```

```
5


Process finished with exit code 0
```

Bileşke Kullanım

İf else kullanımı gibi düşünebiliriz. Örneğin let ve run kullanımında eğer değişken değeri boş değil ise let ile eğer değişken değeri bol ise run ile işlem yapabiliriz.

Örnek:

```
fun main() {
    var isim:String?= null
    isim="Beyza"
    isim?.let{
       println("let çalıştı isim :"+isim)
    }?:run{
      println("run çalıştı : "+ isim)
    }
}
```

```
let çalıştı isim :Beyza

Process finished with exit code 0
```

Örnek:

```
fun main() {
    var isim:String?= null

    isim?.let{
       println("let çalıştı isim :"+isim)
    }?:run{
      println("run çalıştı : "+ isim)
    }
}
```

```
run çalıştı : null

Process finished with exit code 0
```

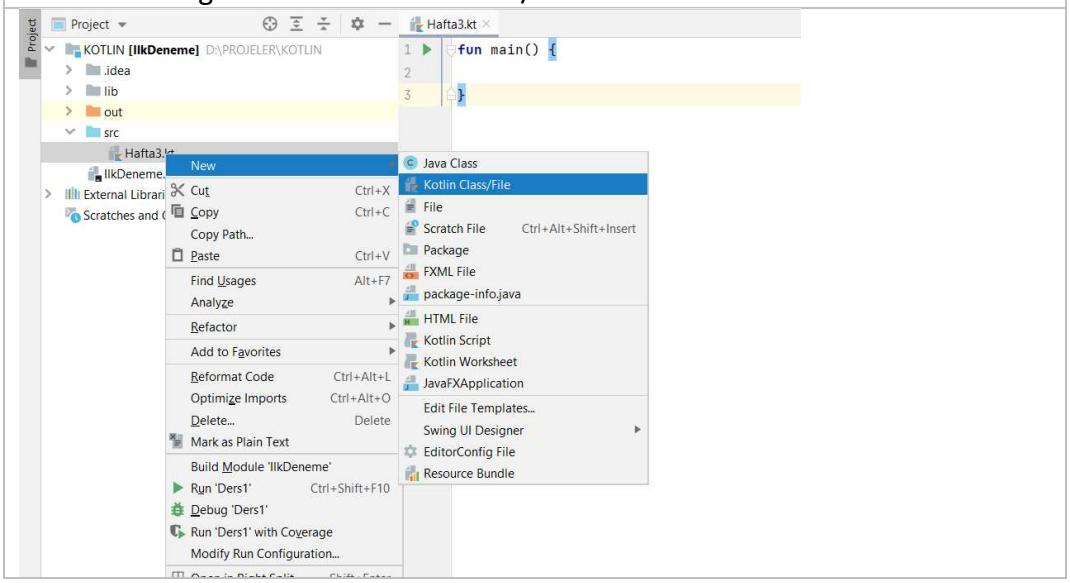
Kotlin Companion Object

Kotlinde static properties ve fonksiyon oluşturamayız fakat companion object oluşturabiliriz.

Bu objenin içine sınıfımızın içinde static olmasını istediğimiz properitsleri, constantları, fonksiyonları tanımlayabiliriz. Javada kullandığımız statikler yerine artık object sınıfları var. Object içine yazdıklarımıza direkt erişim sağlanır.

MyKotlinUtils adında bir kotlin claası oluşturalım

Hafta3.kt > sağ tık > New > Kotlin Class/File



Hafta3.kt	MyKotlinUtils.kt

`fun main() {`
`println(MyKotlinUtils.BASE_URL)`
`}`
	`object MyKotlinUtils {`
	`var BASE_URL="HTTPS..."`
	`}`

```
HTTPS...

Process finished with exit code 0
```

Kotlin Data Class

Data class bir class tanımlamasıdır, aşağıdaki fonksiyoneliteleri çok kısa bir şekilde içine alır(otomatik tanımlar):

- constructor
- fields
- getter ve setter fonksiyonları
- hashCode(), equels() ve toString() fonksiyonları

javada örneğin Ogrenci adında bir sınıfta oluşturduklarımız constructor,getter ve setter dan oluşan kod:

Ogrenci.java

```
public class Ogrenci {
    String adi;
    String soyadi;
    int yasi;

    public Ogrenci(){

    }

    public String getAdi() {
        return adi;
    }

    public void setAdi(String adi) {
        this.adi = adi;
    }

    public String getSoyadi() {
        return soyadi;
    }

    public void setSoyadi(String soyadi) {
        this.soyadi = soyadi;
    }

    public int getYasi() {
        return yasi;
    }

    public void setYasi(int yasi) {
        this.yasi = yasi;
    }

    public Ogrenci(String adi, String soyadi, int yasi) {
        this.adi = adi;
        this.soyadi = soyadi;
        this.yasi = yasi;
    }




    @Override
    public String toString() {
        return "Ogrenci{" +
                "adi='" + adi + '\'' +
                ", soyadi='" + soyadi + '\'' +
                ", yasi=" + yasi +
                '}';
    }
}
```

Javada bu kadar yazdığımız kod , kotlinde :

```
data class Ogrenci(val adi: String, val soyadi:String, var yasi: Int)
```

şeklinde tanımlanır.

Örnek:

```
fun main() {

    var ogrenci=Ogrenci(
            adi = "Beyza",
            soyadi="Koyulmus",
            yasi=37
    )

    println(ogrenci)
}
data class Ogrenci(val adi: String, val soyadi:String, val yasi:Int)
```

```
Ogrenci(adi=Beyza, soyadi=Koyulmus, yasi=37)

Process finished with exit code 0
```

- Boş değer gelebilmesini istediğimizde örnekteki gibi tanımlama yapabiliriz.

Örnek:

```
fun main() {

    var ogrenci2=Ogrenci(
            adi = "Ali",
            soyadi=null,
            yasi=null
    )
    println(ogrenci2)
}
data class Ogrenci(val adi: String?, val soyadi:String?, val yasi:Int?)
```

```
Ogrenci(adi=Ali, soyadi=null, yasi=null)

Process finished with exit code 0
```

Operatörler

Operatörler, program içinde karşılaştırma, matematiksel ve mantıksal işlemlerin yapılmasını sağlayan sembollerdir.

Bütün programlama dillerinde bulunan operatörler ile bir program içerisinde istenilen işlemleri yapabiliriz.

Aritmetik Operatörler

Program içerisinde matematiksel işlemler yapabilmemizi sağlayan operatörlerdir.

Aritmetik operatörleri kullanarak toplama, çıkarma, çarpma, bölme ve mod işlemlerini yapabiliriz.

12 + 6 = 18

12 – 6 = 6

12 * 6 = 72

12 / 6 = 2

12 % 2 = 0

8 % 3 = 2 Modüler Aritmetik bir sayıyı böldüğümüzde kalanı verir.

Atama Operatörleri

Temel kullanım amacı, değişkene değer atanması veya bir başka değişkendeki değeri diğer bir değişkene atamayı sağlamaktır.

Örnek:

```
fun main() {

    var num1 : Float = 11f
     var num2 :Float = 4f

     num2+=num1 // num2=num2+num1
     println("num2 : $num2")
}
```

```
num2 : 15.0


Process finished with exit code 0
```

Karşılaştırma ve Eşitlik Operatörleri

Değişken veya veriler arasında karşılaştırma yapmak istenildiğinde kullanılan operatörlerdir.

>	Büyüktür	a > b	a.compareTo(b)>0
≥	Büyük eşittir	a>=b	a.compareTo(b)>=0
<	Küçüktür	a < b	a.composeTo(b) < 0
≤	Küçük eşittir	a<=b	a.composeTo(b) <=0
==	Eşit eşittir	a==b	a?.equals(b) ? : (b==null)
!=	Eşit değildir	a!=b	!(a?.equals(b) ?:(b==null)

Mantıksal Operatörler

Program içerisinde mantıksal işlemler yapılmasını sağlayan operatörlerdir.

Sıklıkla karşılaştırma operatörüyle beraber kullanılır.

Mantıksal operatörler hem yazılımda hem de lojik devrelerde kullanılır.

AND (ve) Operatörü

Bu operatör kullanıldığında yapılan işlemin sonucu **true** çıkması için tüm verilerin true olması gerekir.

VE operatörü Kotlin dilinde **&&** karakteri ile temsil eder

1.Değer	2.Değer	Sonuç
false	false	false
true	false	false
false	true	false
true	true	true

OR (Veya)Operatörü

Bu operatör kullanıldığında yapılan işlemin sonucu true çıkması için tek bir verinin true olması gerekir.

VEYA operatörü Kotlin dilinde || karakteri ile temsil edilir.

1.Değer	2.Değer	Sonuç
false	false	false
true	false	true
false	true	true
true	true	true

NOT (DEĞİL)Operatörü

Mantıksal Değil operatörü boolean tipinde bir sonucun tersini vermektedir.

DEĞİL operatöründe gelen veri false ise true, true ise false üretir.

Kotlin' de **DEĞİL** operatörü **!** karakteri ile temsil edilir.

Değer	Sonuç
True	False
False	True

6:2(2+1)= sonucu 1 mi ? , 9 mu?

Bilgisayar biliminde hesap makinesini geliştirirken, eğer kararsız kalınırsa yani iki tane işlem önceliği aynı üstünlükteyse soldan sağa doğru yapılmasına karar verilmiştir. 1973 ten önce bu sorunun cevabı 1 iken, 1973'ten sonra cevap bilgisayar biliminde 9 olarak kabul edilmiştir. Matematik bilimi de sanal hayata uymak zorunda kalıp cevabı 9 olarak kabul etmektedir.

Bu gibi durumlarda işlem kararsızlığını önlemek için önceliklere dikkat etmemiz gerekir.

İşlem Önceliği

1. () pazantez içi
2. / veya * (bölme veya çarpma)
3. % mod alma
4. + veya – (toplama veya çıkarma)

İf-else if – else (if yapısı)

Günlük hayatımızda bizler, birtakım koşullara bağlı olarak kararlar veririz. Örneğin dışarıda yağmur yağıyorsa şemsiye almamız daha iyidir. Bilgisayarların sağladığımız koşullara ve girdilere dayalı kararlar vermesini istiyoruz. Tam da bu noktada if-else yapısı devreye giriyor.

Bu bir kontrol akışı ifadesidir, yani koşullara göre kodun akışına karar verir. Bunlara koşullu ifadeler denir. Aslında programa basitçe şu şekilde diyoruz:

"Bu koşulun doğru olup olmadığını kontrol et; evet ise X işlemini yap, yoksa Y işlemini yap."

Burada dikkat etmemiz gereken, doğru talimatları girmektir.

Özetle **if** İngilizce' de **"eğer",** else de **"değilse"** demektir. Eğer böyle ise şöyle şöyle yap tarzındaki koşulları bilgisayara anlatmak için kullanılır.

if - else yapısı:	

	`if (koşul){`
	`  //koşul sağlanıyorsa bu kod bloğu yürütülür.`
	`}`
	`else {`
	`  //koşul sağlanmıyorsa bu kod bloğu yürütülür.`
	`}`

- Şart/koşul doğru ise durum 1 yanlış ise durum 2 gerçekleşecektir.
- Eğer şart yerine 0 sayısı yazılırsa veya gelirse şart sağlanmamış,
- Şart yerine 0 harici bir sayı gelirse veya yazılırsa şart sağlanmış demektir.
- Else her zaman yazılmak zorunda değildir, aksi bir durum incelenmek istenirse yazılır.
- Boolean, yürütülen koşul koduna bağlı olarak doğru veya yanlış olabilir.

Örnek: Bir sayının tek mi çift mi olduğunu kontrol eden bir program yazalım.

```
fun main() {
   var sayi = 7

    if(sayi%2==0)
    {
        println("Sayi çifttir")
    }
    else
    {
        println("Sayi tektir")
    }
}
```

```
/*Eğer sayı ikiye bölündüğünde kalan sıfır olursa, sayımız çifttir.
Değilse sayı tek olacaktır. Burada sayı değeri 7 olarak verilmiş. İf
şartında sayiyi yani 7'yi ikiye böldüğümüzde kalan sıfır olsaydı "sayi
çifttir" yazacaktı. İlk şart sağlanmadığı için else yapısının kod kısmına
geçer ve "Sayi tektir" yazdırır.*/
```

- Birden fazla if-else koşulumuzda olabilir. Bu defa if-else if -else yapısını kullanırız.

if – else if – else yapısı:	

	`if (koşul1) {`
	`  //koşul1 sağlanıyorsa bu kod bloğu yürütülür. Diğer bloklara girmez.`
	`}`
	`else if (koşul2) {`
	`  //koşul1 sağlanmayıp koşul2 sağlanıyorsa bu kod bloğu yürütülür.`
	`}`
	`else if (koşul3) {`
	`  //koşul1 ve koşul2 sağlanmayıp koşul3 sağlanıyorsa bu kod bloğu yürütülür.`
	`}`
	`… //İstediğimiz kadar else if yazabiliriz.`
	`else {`
	`  //yukarıdaki koşulların hiçbiri sağlanmıyorsa bu kod bloğu yürütülür.`
	`}`

Örnek:

```
fun main() {

   val a = 13
   val b = 2



    if(a>b)
    {
        println("a sayisi b sayısından büyüktür.")

    }
    else if(a<b)
    {
        println("a sayisi b sayisindan küçüktür.")
    }
    else
    {
        println("a sayisi b sayisina eşittir")
    }
```

```
/* a sayısı 13 ve b sayısı 2 olarak verilmiş. İlk if şartında a>b yani 13
>2 şartı sağlandığı için ilk kod bloğuna girecek ve ekrana "a sayisi b
sayisından büyüktür." yazdırılacak.*/
```

Örnek:

```
fun main() {

   var sayi = 5

    if(sayi>5)
    {
        println("Sayi 5'den büyük")

    }
    else if(sayi<5)
    {
        println("Sayı 5'den küçük")
    }
    else
    {
        println("Sayı 5 e eşit")
    }
}
```

```
Sayı 5 e eşit


Process finished with exit code 0
```

- İf – else deyimlerini iç içe de kullanabiliriz. Herhangi bir kod bloğunun içinde kullanılabilir.

```
fun main() {

    if(koşul1){
           if(koşul2){ //işlemler  }
           else{ //işlemler  }
    }
    else
    { }
}
```

```
/* Buradaki yapıda eğer ilk koşul sağlanırsa ilk blok içine girilecek.
Burada tekrar bir if yapısı mevcut. Duruma göre yani içerdeki koşul
sağlanıyorsa iç if bloğu çalışacak değilse içerideki else bloğu
çalışacak.

Eğer ilk koşulda şart sağlanmazsa dışarıdaki else yapısı çalışacak ve
program sonlanacaktır.*/
```

- Kotlin'de bir değişkene if – else işleminin sonucunu atayabiliriz. Örneğin bir sayının çift mi tek mi olduğunu if – else ifadesi kullanarak değerlendiren bir program yazalım.

```
fun main() {

   val sayi=12
   val sonuc= if(sayi%2==0){
                            "sayi çifttir."
                            }
              else){
                     "sayi tektir."
                    }
   println(sonuc)
}
```

```
/*İstersek süslü parantezleri kaldırabiliriz. O zaman kod şu şekilde
olacaktır :

   val sonuc= if(sayi%2==0)"Çift" else "Tek"
```

Elvis Operatörü

? : sembolleriyle kullanılır. Sol taraftaki durum geçerliyse değişkene sol kısmı atar. Değilse sağ taraftaki kısmı değişkene atar.

Örnek:

```
fun main() {

    val liste = mutableListOf<Int>(1,2,3)

    val boyut =liste?.size?:-1
    //liste boş olmadığı için liste boyutu, boyut isimli değişkene atandı

    println(boyut)
}
```

```
3

Process finished with exit code 0
```

When Expression

- When yapısı Java'daki switch - case yapısına karşılık gelmektedir.
- if – else ile yapılabilen her şey when yapısı ile yapılabilir. İf- else göre daha kolay bir kullanımı vardır.
- When yapısı, Kotlin' de koşullu ifade yazmanın başka bir yoludur.
- Birden çok koşulumuz olduğunda bunu bir if – else ifadesi kullanarak yazmak hem zahmetli hem de daha az okunurdur. Okunabilirliği artırmak için when yapısını kullanabiliriz. Birkaç örnek verelim.

Örnek:

```
fun main() {

    val not= "CB"
    if (not== "AA"){
    println("Harf notu AA dır.")

   }
    else if (not== "BA"){
    println("Harf notu BA dır.")

   }
    else if (not== "BB"){
    println("Harf notu BB dir.")

   }
   else if (not== "CB"){
    println("Harf notu CB dir.")

   }
   else if (not== "CC"){
    println("Harf notu BB dir.")

   }
   else if (not== "DD"){
    println("Harf notu BB dir.")
```

```
    }
    else {
     println("Harf notu FF dir.")

    }

}
```

```
// Bu şekilde bir çok şart yapımız oluşuyor ve kod satırı uzun bir hal alıyor. Okunabilirliğini artırmak için bunun yerine when yapısını kullnabiliriz :
```

```
fun main() {

  val not= "CB"

  when(not){

  "AA"-> println("Harf notu AA dır.")
  "BA"-> println("Harf notu BA dır.")
  "BB"-> println("Harf notu BB dır.")
  "CB"-> println("Harf notu CB dır.")
  "CC"-> println("Harf notu CC dır.")
  "DD"-> println("Harf notu DD dır.")
  else -> println("Harf notu FF dır.")

 }

}
```

- Her iki kod da aynı işlemi yapmaktadır. Yazdığımız koda göre hangisini kullanacağımıza karar verebiliriz.

- İf – else yapısında olduğu gibi bir değişkene when yapısını kullanarak değer ataması yapabiliriz. Az önceki örnek üzerinde yazalım:

```
fun main() {

  val not= "CB"

  val sonuc = when(not){

  "AA"-> println("Harf notu AA dır.")
  "BA"-> println("Harf notu BA dır.")
  "BB"-> println("Harf notu BB dır.")
  "CB"-> println("Harf notu CB dır.")
  "CC"-> println("Harf notu CC dır.")
  "DD"-> println("Harf notu DD dır.")
  else -> println("Harf notu FF dır.")
  }

 println(sonuc)

}
```

Örnek:

```
fun main() {

    val sayi =0
    when(sayi)
    {
        2 ->
        {
            println("Sayi 2 dir ")
        }
        8 ->
        {
            println("Sayi 8 dir ")
        }
        else ->
        {
            println("Sayi 2 ve 8 değildir.  ")
        }
    }
}
```

```
Sayi 2 ve 8 değildir.

Process finished with exit code 0
```

Örnek:

```
fun main() {

    val yas =14

    val sonuç= when(yas)
    {

      11 -> "Onbir"

      12 -> " Oniki"

      13..19 ->"Genç"
     // yaş 13 - 19 aralığında ise "Genç"

     }

    println(sonuc)
}
```

Enums (Enumlar)

Enum sınıflar, Enumaration yani numaralandırmanın yazılımda karşılık bulmuş halidir. Belirli değerlere karşılık gelen tanımlamalar için kullandığımız bir yapıdır.

Bu sınıflar sadece Kotlin diline özgü olmayıp C++, C #, Java gibi diğer birçok programlama dillerinde de sıkça karşılaşabileceğimiz yapılardır.

Enumların var olma nedeni Clean Code (Temiz Kod) yazmak, yani daha okunaklı kod yazmak içindir.

```
enum class USER_TYPE(val get: Int)
{
    ADMIN(1),
    USER(2),
    SUPER_ADMIN(3)
}
```

Kotlin' de Hata Ayıklama

Exceptionlar, programın çalışma zamanında meydana gelebilecek ve programımızı aniden sonlandırabilecek istenmeyen sorunlardır. Hata ayıklama, kodu bu tür exceptionlardan koruyabileceğimiz bir işlemdir. Try – catch anahtar kelimelerini kullanarak haya ayıklama yapısını oluşturabiliriz.

Try – Cath Kullanımı :	

	`try {`
	`  //Hata yakalanacak kod`
	`}`
	`catch (e: ArithmeticException) {`
	`  //Hatayı handle etme`
	`  //yazılmak istenen mesaj yazılabilir`
	`}`
	`catch (e: Exception) {`
	`  //Hatayı handle etme`
	`  //yazılmak istenen mesaj yazılabilir`
	`}`
	`finally {`
	`  //her zaman bu skop çalışır`
	`}`

`try` bloğu içerisinde asıl yapacağımız işlemler yer alır. Burada olası bir hata oluştuğunda, hata hangi **`catch`** içerisinde tanımlıysa o bloğun içine girer ve gerekli hata mesajı yazdırılır. İşlemlerde herhangi bir hata yoksa catch blokları atlanır ve **`finally`** bloğundan devam edilir.

`finally` bloğu, hata olsa da olmasa da çalışan bloklardır.

Örnek:

```
  fun main() {

      try{
          var sayi =10/0
          println("İslem yapılıyor")
          println(sayi)
         }

      catch (e : ArithmeticException ){

          println("ArithmeticException")

         }
      finally{
          println("finally bloğu her zaman çalışır")
      }
}
```

```
ArithmeticException
finally bloğu her zaman çalışır
```

- Bir sayıyı sıfıra bölmek mümkün değildir, tanımsızdır. Dolayısıyla **`try`** bloğu içerisinde bir hata oluşur ve try bloğunun içerisindeki her şeyi yok sayıp **`catch`** bloğuna geçer. Burada istenen yapılır. En son **`finally`** bloğu da yürütülüp program sona erer.

`try - catch` yapısını kullanmazsak programımız aşağıda yazdığımız gibi aniden bir hata ile sonlanırdı.

```
fun main(){

          var sayi =10/0
          println("İslem yapılıyor")
          println(sayi)

         }
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
   at FileKt.main (File.kt:3)
   at FileKt.main (File.kt:-1)
   at sun.reflect.NativeMethodAccessorImpl.invoke0 (NativeMethodAccessorImpl.java:-2)
```

Çok Boyutlu Diziler (Matrisler)

Daha önceki bölümlerde programlama dilinde tek boyutlu dizileri anlatmıştık. Tek boyutlu diziler tek bir diziden meydana gelir. Bu diziler tek bir satır ve en az bir sütun içerirler. Örneğin [3] ; satır ve sütun sayısı bir olan tek boyutlu dizidir. Mesela [1, 3, 5] ; satır sayısı bir, sütun sayısı üç olan tek boyutlu bir dizidir. Tek boyutlu bir dizi yalnızca bir satırdan oluşur. Satır sayısı arttırıldığında çok boyutlu diziler meydana gelir. Böylece bir dizinin daha fazla eleman tutması sağlanır.

Örneğin 3x3'lük bir matris yazmak istediğimizde,her bir satır ayrı ayrı düşünüldüğünde aşağıdaki gibi yazabilirdik.

$A=\begin{bmatrix}[1 & 2 & 3] \\ [4 & 5 & 6] \\ [7 & 8 & 9]\end{bmatrix}$	`val a1=intArrayOf(1, 2, 3)`
	`val a2=intArrayOf(4, 5, 6)`
	`val a3=intArrayOf(7, 8 ,9)`

Bunu bütün bir matris olarak yazmak istediğimizde aşağıda yazdığımız kod gibi başka bir Array içerisine alıyoruz.

$A=\begin{bmatrix}1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9\end{bmatrix}$	`import java.util.*`
	
	`fun main(args: Array<String>) {`
	
	`val A =arrayOf(`
	`arrayOf(1, 2, 3),`
	`arrayOf(4, 5, 6),`
	`arrayOf(7, 8, 9)`
	`)`
	`println(Arrays.deepToString(A))`
	`}`

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]


Process finished with exit code 0
```

Döngü Kavramı

Döngüler tüm programlama dillerinde olan temel programcılık kavramlarındandır.

Döngüleri, tekrar edilen işlemlerde kullanmak bize oldukça kolaylık sağlar.

Repeat Döngüsü

Repeat döngüsünü örnek üzerinde inceleyelim.

Örnek: repeat döngüsü kullanarak 0 dan 10 kadar olan sayıları alt alta yazdıralım.

```
fun main() {
    repeat(10){
        println(it)
    }
}
```

```
0
1
2
3
4
5
6
7
8
9

Process finished with exit code 0
```

Örnek: repeat döngüsü kullanarak 0 dan 10 kadar olan sayıları alt alta yazdıralım.

```
fun main() {
     var toplam =0
     repeat(10){
     println(it)
     toplam+=it
    }
    println("toplam :" + toplam)
}
```

```
1
2
3
4
5
6
7
8
9
toplam :45

Process finished with exit code 0
```

For Döngüsü

- Belli aralıklarla tekrar edilen işlemleri basit bir şekilde yapmak için kullanılır.
- For döngüsünde **in** operatörü kullanılarak döngü kurulur.
- Döngüdeki x değişkeni, koleksiyona ait her bir değere karşılık gelmektedir.
- Döngü her döndüğünde koleksiyona ait değerler tek tek değişkene yazdırılır.

- Döngüde belirtilen koleksiyon alanına: dizi, liste, range(aralık) ve string bir değer gelebilir.

Döngünün kullanımı şu şekildedir:	

	`for (x in koleksiyon){`
	`  //yapılacak işlem`
	`}`

For Döngüsü (Range)

Örnek: For döngüsünü kullanarak 1 den 10 a kadar olan sayıları yazalım.

```
fun main() {

    for (x in 1..10){

        println("x : "+x)

    }
}
```

```
x : 1
x : 2
x : 3
x : 4
x : 5
x : 6
x : 7
x : 8
x : 9
x : 10

Process finished with exit code 0
```

For Döngüsü (Range Step)

Örnek: 1 den başlayıp ikişer artırarak 10 kadar yazalım.

```
fun main() {

    for (x in 1..10 step 2){

        println("x : "+x)

    }
}
```

```
x : 1
x : 3
x : 5
x : 7
x : 9

Process finished with exit code 0
```

For Döngüsü (downTo Step)

Geriye doğru azalarak oluşan bir döngü oluşturmak istediğimizde **downTo** kullanmamız gerekir.

Örnek: 10 dan başlayıp birer birer azalarak 1 e kadar yazalım.

```
fun main() {

    for (x in 10 downTo 1){

        println("x : "+x)

    }
}
```

```
x : 10
x : 9
x : 8
x : 7
x : 6
x : 5
x : 4
x : 3
x : 2
x : 1

Process finished with exit code 0
```

Örnek: 20 den başlayıp üçer üçer azalarak 1 e kadar yazalım.

```
fun main() {

    for (x in 20 downTo 1 step 3){

        println("x : "+x)

    }
}
```

```
x : 20
x : 17
x : 14
x : 11
x : 8
x : 5
x : 2

Process finished with exit code 0
```

For Döngüsü (Until)

Until kullanarak yazdığımız döndülerde -e kadar işlem yapılması istenir. Yani son yazılan sayı için işlem yapılmaz. Until kullanarak 1 den 10'a kadar yazılmasını istediğimizde 10 dahil edilmez, döngü 10'a gelince biter, dolayısıyla 9 a kadar yazar.

Örnek: Until kullanarak 1'den 10'a kadar yazalım

```
fun main() {
    for (x in 1 until 10) {
        println("x : "+x)
    }
}
```

```
x : 1
x : 2
x : 3
x : 4
x : 5
x : 6
x : 7
x : 8
x : 9

Process finished with exit code 0
```

For Döngüsü (Until Step)

Örnek:

```
fun main() {
    for (x in 0 until 12 step 3) {
        println("x : "+x)
    }
}
```

```
x : 0
x : 3
x : 6
x : 9

Process finished with exit code 0
```

Örnekte görüldüğü gibi until kullanıldığı için 12 sayısını yazmadı, çünkü 12.döngüye geldiğinde işlem bitti. 12 yi de yazılmasını istiyorsak döngüyü 13'e kadar yapabilir ya da until yerine aralık olarak belirtebiliriz. (0..12) gibi.

For Döngüsü (String)

Örnek:

```
fun main() {
    for (x in "Medipol") {
        println("x : "+x)
    }
}
```

```
x : M
x : e
x : d
x : i
x : p
x : o
x : l

Process finished with exit code 0
```

Örnek:

```
fun main() {

    val isim ="Medipol"

    for (x:Char in isim ) {
        println("x : "+x)
    }
}
```

```
x : M
x : e
x : d
x : i
x : p
x : o
x : l

Process finished with exit code 0
```

For Döngüsü (Array)

Örnek:

```
fun main() {

    val isimler =arrayOf("Beyza", "Hamza","Tuğba","Görkem","Gizem")
    for(x in isimler)
    {
        println("isim :" + x)
    }
}
```

```
isim :Beyza
isim :Hamza
isim :Tuğba
isim :Görkem
isim :Gizem

Process finished with exit code 0
```

For Döngüsü (Koleksiyon)

Örnek:

```
fun main() {

    val isimler =listOf("Beyza", "Hamza","Tuğba","Görkem","Gizem")
    for(x in isimler)
    {
        println("isim :" + x)
    }
}
```

```
isim :Beyza
isim :Hamza
isim :Tuğba
isim :Görkem
isim :Gizem

Process finished with exit code 0
```

For Döngüsü (index)

Listedeki elemanların indexleriyle birlikte almak için withIndex() kullanılır.

Örnek:

```
fun main() {

    val isimler =listOf("Beyza", "Hamza","Tuğba","Görkem","Gizem")

    for(x in isimler.withIndex())
    {
        println("isim :" + x)
    }
}
```

```
isim :IndexedValue(index=0, value=Beyza)
isim :IndexedValue(index=1, value=Hamza)
isim :IndexedValue(index=2, value=Tuğba)
isim :IndexedValue(index=3, value=Görkem)
isim :IndexedValue(index=4, value=Gizem)

Process finished with exit code 0
```

Listedeki indexleri ayrı değerleri ayrı olarak almak istediğimizde şu şekilde alabiliriz:

Örnek:

```
fun main() {

    val isimler =listOf("Beyza", "Hamza","Tuğba","Görkem","Gizem")

    for(x in isimler.withIndex())
    {
        println("index : "+x.index.toString()+"  isim :" + x.value)
    }
}
```

```
index : 0  isim :Beyza
index : 1  isim :Hamza
index : 2  isim :Tuğba
index : 3  isim :Görkem
index : 4  isim :Gizem

Process finished with exit code 0
```

- For döngüsü index kullanımı şu şekilde de olabilir :

```
for((index,eleman) in isimler.withIndex())
```

- index ve eleman yerine herhangi bir şey yazabiliriz.

Örnek:

```
fun main() {

    val isimler =listOf("Beyza", "Hamza","Tuğba","Görkem","Gizem")

    for((i,v) in isimler.withIndex())
    {
        println(" " + i +" . isim " +v)
    }
}
```

```
0 . isim Beyza
1 . isim Hamza
2 . isim Tuğba
3 . isim Görkem
4 . isim Gizem

Process finished with exit code 0
```

For Döngüsü (Index Null)

Listelerde null kullanılabiliyordu. Bu yüzden null olan değeri de yazdırabiliriz.

Örnek:

```
fun main() {

    val isimler =listOf("Beyza", "Hamza","Tuğba","Görkem","Beyza",null)

    for((i,v) in isimler.withIndex())
    {
        println(" " + i +" . isim " +v)
    }
}
```

```
0 . isim Beyza
1 . isim Hamza
2 . isim Tuğba
3 . isim Görkem
4 . isim Beyza
5 . isim null

Process finished with exit code 0
```

Buraya kadar olan döngüde listeyi kullandık. Bundan sonraki örneklerde Set, yani küme örneklerine bakacağız.

For Döngüsü (Index Set)

setOf kullanırken dikkat etmemiz gereken bir eleman ile yalnızca bir işlem yapılabilir. Aynı elemandan birden fazla olduğu durumda sadece bir eleman yazılır.

Örnek:

```
fun main() {

    val isimler =setOf("Beyza", "Hamza","Tuğba","Görkem","Beyza",null)

    for(x in isimler.withIndex())
    {
        println( x.index.toString()+". x: " +x.value)
    }
}
```

```
0. x: Beyza
1. x: Hamza
2. x: Tuğba
3. x: Görkem
4. x: null

Process finished with exit code 0
```

SetOf içerisinde iki tane Beyza olmasına rağmen sadece bir tanesini yazdırır.

ForEach Döngüsü

Örnek:

```
fun main() {

    (1..10).forEach{ x->

        println("x : " +x)

    }

}
```

```
x : 1
x : 2
x : 3
x : 4
x : 5
x : 6
x : 7
x : 8
x : 9
x : 10

Process finished with exit code 0
```

ForEach Döngüsü (Step)

Örnek:

```
fun main() {

    (1..10 step 2).forEach{ x->

        println("x : " +x)

    }
}
```

```
x : 1
x : 3
x : 5
x : 7
x : 9

Process finished with exit code 0
```

ForEach Döngüsü (Koleksiyon)

Örnek:

```
fun main() {

val isimler= mutableListOf("Beyza", "Hamza", "Ayşe")
    isimler.forEach{ x->

        println("x : " +x)

    }
}
```

```
x : Beyza
x : Hamza
x : Ayşe

Process finished with exit code 0
```

Break Kavramı

Break anahtar kelimesi ile o an ki kod bloğundan çıkmasını sağlar.

Örnek:

```
fun main() {

    for (x in 1..10)
    {
        println(" x : " +x)

        if(x==3)
        {break}
    }
}
```

```
x : 1
x : 2
x : 3

Process finished with exit code 0
```

Continue Kavramı

Continue anahtar kelimesi break kavramı ile aynı çalışır. Tek farkı döngüyü sonlandırmaz, sadece o anki durumunu atlar.

Örnek:

```
fun main() {

    for (x in 1..7)
    {
        if(x==3)
        {
            continue
        }
        println(" x : " +x)
    }
}
```

```
x : 1
x : 2
x : 4
x : 5
x : 6
x : 7

Process finished with exit code 0
```

Örnek: İç içe döngü kullanarak çarpım tablosu oluşturalım.

```
fun main() {

    for (i in 1..10)
    {
        for(j in 1..10)
        {
            println("$i x $j = "+(i*j))
        }
        println("*****************")
    }
}
```

```
"C:\Program Fi
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
1 x 10 = 10
**************
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
**************
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
.
.
.
8 x 9 = 72
8 x 10 = 80
*****************
9 x 1 = 9
9 x 2 = 18
9 x 3 = 27
9 x 4 = 36
9 x 5 = 45
9 x 6 = 54
9 x 7 = 63
9 x 8 = 72
9 x 9 = 81
9 x 10 = 90
*****************
10 x 1 = 10
10 x 2 = 20
10 x 3 = 30
10 x 4 = 40
10 x 5 = 50
10 x 6 = 60
10 x 7 = 70
10 x 8 = 80
10 x 9 = 90
10 x 10 = 100
*****************

Process finished
```

Örnek: A = 1 *e* 3 2 5 4 olmak üzere bu matrisi döngü kullanarak yazdıralım.

```
fun main() {

    val A= arrayOf(
            arrayOf(1, "e", 3),
            arrayOf(2, 5, 4)
    )
    for (i in 0 until A.size) //A nın 2 elemanı vardır.İçerisinde 2 dizi vardır.
    {
        for(j in 0 until A[i].size)
        {
            print(""+(A[i][j]) +" ")
        }
        println(" ")
    }
}
```

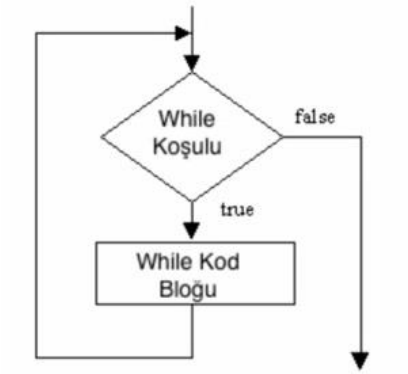
```
1 e 3
2 5 4

Process finished with exit code 0
```

While Döngüsü

While yapısı koşulun sonucu true olduğu sürece kod bloğundan çıkmaz ve sürekli kod bloğunun içindeki kodu çalıştırmaya devam eder.

For döngüsündeki kolaylıklara sahip değildir.

While döngüsü kullanım şekli :




```
while("Koşul")
{

   //Koşulun sağlandığı zaman çalışacak kod bloğu

}

 //Koşul sağlanmazsa eğer döndüden çıkar
```

Örnek: While döngüsü kullanarak sıfırdan 10'a kadar sayıları yazdıralım.

```
fun main() {

   var i=0
  while(i<11)

  {  //i 11 den küçük olduğu müddetçe true dönecektir ve while kod
bloğuna girecektir.
     print("    i ="+i)
     i++  //her defasında i'yi 1 artırır.
  }

}
```

```
    i =0    i =1    i =2    i =3    i =4    i =5    i =6    i =7    i =8    i =9    i =10
 Process finished with exit code 0
```

Do While Döngüsü

Do while döngüsü, while döngüsüne benzerdir fakat burada önce işlem yapılır daha sonra şarta bakılır.

- do while döngüsünde şart sağlansa da sağlamansa da kod bloğuna en az bir kez girer.

Örnek:

do while	while

`fun main() {`
 	
`   var i=11`
 	
`   do`
`  {`
`     print("    i ="+i)`
`//önce yazdırma işlemi yapacak daha sonra şarta bakılacak`
`     i++`
`  //her defasında i'yi 1 artırır.`
`  } while(i<11)`
`//i 11 den küçük olduğu müddetçe true dönecektir ve while kod bloğuna girecektir.`
 	
`}`
	`fun main() {`
	
	`   var i=11`
	
	`   while(i<11)`
	`//i 11 den küçük olduğu müddetçe true dönecektir ve while kod bloğuna girecektir.`
	`  {`
	`     print("    i ="+i)`
	`//önce yazdırma işlemi yapacak daha sonra şarta bakılacak`
	`     i++`
	`//her defasında i'yi 1 artırır.`
	`  }`
	
	`}`

`   i =11` `Process finished with exit code 0`	`Process finished with exit code 0`

Döngülere İsim Vermek

@ işareti ile döngülere isim verebiliriz. İç içe döngülerde bazen içerdeki döngüdeki şart sebebiyle dışarıdaki döngüyü kırmamız gerekebiliyor.

Örnek:

```
fun main() {

    carpan@ for(i in 1..5){
        carpilan@ for(j in 1..5){

            println("$i x $j =" + (i*j))

            if(j==3)

            {
              break@carpan
            }
        }
    }

}
```

```
1 x 1 =1
1 x 2 =2
1 x 3 =3

Process finished with exit code 0
```

Örnek:

```
fun main() {

    carpan@ for(i in 1..5){
        println("**********")
        carpilan@ for(j in 1..5){

            println("$i x $j =" +
(i*j))

            if(j==3)
            { continue@carpan}
        }
    }

}
```

```
*********
1 x 1 =1
1 x 2 =2
1 x 3 =3
*********
2 x 1 =2
2 x 2 =4
2 x 3 =6
*********
3 x 1 =3
3 x 2 =6
3 x 3 =9
*********
4 x 1 =4
4 x 2 =8
4 x 3 =12
*********
5 x 1 =5
5 x 2 =10
5 x 3 =15
```

Random Kullanımı

Örnek: 0 ile 10 arasında rastgele bir sayı üretip yazalım.

```
fun main() {

    val rastgele = (0..10).random()
    print(rastgele)

}
//0 ile 10 arasında rastgele sayı
üretir.
Kodu her çalıştırdığımızda farklı
bir sayı gelecektir
```

```
5
Process finished with exit code 0
```

Örnek:

```
fun main() {

    var rastgele =0

    while(rastgele!=10 )
    {
        rastgele=(0..10).random()
        println(rastgele)
    }

}
```

```
//ilk çalıştırdığımızda
4
10

Process finished with exit code 0
```

```
//ikinci çalıştırdığımızda
5
9
7
5
10

Process finished with exit code 0
```

Klavyeden Girilen Metnin Boyutunu Bulma

Örnek:

```
fun main() {
    var klavye = readLine() // kullanıcıdan değer alma
    print("girilen metin boyutu:" +klavye!!.length)

}
```

```
Beyza KOYULMUŞ
girilen metin boyutu:14
Process finished with exit code 0
```

Fonksiyon Kavramı

Programlamada fonksiyonlar, belli kodları bir arada tutup farklı amaçlara göre yönetmeyi ve tekrardan kaçınmayı sağlayan kod bloklarıdır.

Aynı matematikte kullandığımız fonksiyonlar gibi istenilen parametrelere argüman alıp belli çıktılar üretebilir.

Kapsamlı uygulamalarda binlerce kod satırı vardır. Bütün kodları ana metodun (main fonksiyon) içine yazmak kesinlikle iyi bir fikir değildir. Bu kodu ayrı ve iyi yönetilebilecek şekilde parçalara ayırmamız daha iyi olacaktır. Her birinin kendine özgü işlevselliklere sahip yapılarını oluşturacağız. Basit ifadeyle fonksiyon, belirli bir görevi veya işlevi yerine getiren bir kod bloğudur. Ne zaman bu işlevselliğe ihtiyaç duysak, kodda bu işlevi çağırırız. Bu fonksiyonlar girdi alabilir ve girdilere göre bazı işlemler gerçekleştirir ve çıktı sağlar. Bu şekilde fonksiyon kullanmamız kod tekrarını önler ve kodun yeniden kullanılmasına yardımcı olur.

Kotlin'de fonksiyonlar sırasıyla;

- "fun" anahtar kelimesi
- Fonksiyon ismi,
- Fonksiyonun parametre ve parametre tipi
- Fonksiyon dönüş tipi

```
fun isim(parametre : Tip) :Çıktı Tipi {
    return islem
}
```

şeklinde yazılır.

Bir fonksiyonu kullanmak 2 aşamadan oluşur. Önce fonksiyonu tanımlamamız, daha sonra bu fonksiyonu çağırmamız gerekiyor. Daha iyi anlamak için bir örnek üzerinde açıklayalım.

```
fun topla(a : Int, b : Int): Int {
    val toplam= a + b
    return toplam
}
```

Açıklama:

- İki tam sayıyı toplayan, **topla** adında bir fonksiyon yazdık. **`fun`** anahtar kelimesi fonksiyon olduğunu temsil eder. Varsa girdiler parantez içinde belirtilir. Buradaki girdileri a ve b olarak tanımladık.
- Yapmak istediğimiz işlemler süslü parantezler içerisine yazılır.
- İşlem kısmı tamamlandıktan sonra **`return`** anahtar kelimesi ile çıktıyı döndürüyoruz. Örneğin örnekte çıktı olarak toplam ifadesini döndürüyoruz.
- Dönen değerin veri türünü de tanımlamamız gerekiyor. İki tam sayının toplamı da bir tam sayı olduğundan veri türünü giriş parametrelerinden hemen sonra tanımlarız, yani : Int

  Buraya kadar fonksiyonun nasıl tanımlanacağını anlattık. Şimdi yazdığımız bir fonksiyonu nasıl kullanacağız buna bakalım.

```
fun main () {
   val sonuc= topla(3,4)
   println(sonuc)
}
```

```
fun topla(a : Int, b : Int): Int {
    val toplam= a + b
    return toplam
 }
```

Açıklama:

- Bir fonksiyonu kullanmak için girdilerine göre değerler vermeliyiz. Örnekte girdi parametreleri iki tam sayı olduğu için Main Metodunda yazdığımız fonksiyona iki tam sayı veriyoruz. (3 ve 4 sayıları) Eğer girdilerimiz string ifadeler olsaydı, fonksiyonumuzun girdilerine string ifadeler yazmamız gerekecekti.

- Eğer fonksiyonumuz örnekte olduğu gibi bir çıktı döndürürse bunu bir değişkende saklayabiliriz. Örnekte topla(3,4) fonksiyonumuzun çıktısını " sonuc" değişkeninde sakladık.

Örnek: f(2) olarak girildiğinde iki fazlasını veren fonksiyonu yazalım.

```
fun main(Args :Array<String>) {

 println("f fonksiyonunun çıktısı : "+f(2))

}

fun f(x:Int) :Int{

  return x+2
}
```

```
f fonksiyonunun çıktısı : 4

Process finished with exit code 0
```

Örnek: 4 sayısının karesini alan programı fonksiyon kullanarak yazalım.

```
fun main(Args :Array<String>) {

 println("beyza fonksiyonunun çıktısı : "+beyza(4))
}

fun beyza(y:Int) :Int{
    return y*y
}
```

```
beyza fonksiyonunun çıktısı : 16

Process finished with exit code 0
```

Örnek: Kullanıcının girdiği sayının karesini alan fonksiyonu yazalım.

```
fun main(Args :Array<String>) {

    println("Bir sayi giriniz")
    var sayi1:Int?=readLine()?.toInt()

    println("Girilen sayının karesi : "+ carp(sayi1!!))
}

fun carp(sayi1:Int):Int{
    return sayi1*sayi1
}
```

```
Bir sayi giriniz
7
Girilen sayının karesi : 49

Process finished with exit code 0
```

Örnek:

```
fun main(Args :Array<String>) {

    println("Birinci sayıyı giriniz")
    var sayi1:Int?=readLine()?.toInt()

    println("İkinci sayıyı giriniz")
    var sayi2:Int?=readLine()?.toInt()

    println("Girilen sayıların çarpımı  : "+ carp1(sayi1!!, sayi2!!))

    println("Girilen sayıların çarpımı  : "+ carp(sayi1!!, sayi2!!))
}

fun carp1(sayi1:Int , sayi2:Int):Int{
    return sayi1*sayi2
}

fun carp(sayi1:Int , sayi2:Int):Int=sayi1*sayi2
```

```
Birinci sayıyı giriniz
3
İkinci sayıyı giriniz
5
Girilen sayıların çarpımı  : 15
Girilen sayıların çarpımı  : 15

Process finished with exit code 0
```

Unit (Void) Değer Tipi

Herhangi bir değer döndürmeyen fonksiyonlar void yani unit değer tipinde fonksiyonlardır. Örnekteki gibi unit yazarak veya hiçbir şey yazmayarak kullanılabilir.

Örnek:

```
fun main(Args :Array<String>) {

 yazdir("Merhaba Kotlin")

}

fun yazdir(message:Any)
{
//Any dediğimizde her türü kabul
etmektedir.

 print(message)
}
```

```
Merhaba Kotlin
Process finished with exit code 0
```

```
fun main(Args :Array<String>) {

 yazdir("Merhaba Kotlin")

}

fun yazdir(message:Any):Unit
{
    print(message)
}
```

```
Merhaba Kotlin
Process finished with exit code 0
```

Single Expression Fonksiyonlar

Return ettiğimiz bazı fonksiyonları tek satırda yazabiliyoruz. Kotlin'de fonksiyonlarda süslü parantezler iptal edilip sadece sonucun = tarafından atanması ile **single expression functions** (tek ifadeli fonksiyonlar) kullanılabilir.

Örnek:

```
fun carp(sayi1 :Int,sayi2 : Int) : Int
{
    return sayi1*sayi2
}
```

```
fun carp(sayi1 :Int,sayi2 : Int) : Int=sayi1*sayi2
```

- Return yazmak yerine = koyarak işlemi yaptırabiliriz. Yazdığımız iki fonksiyonda aynı işlemi yapmaktadır.

Default Arguments

Fonksiyonlardaki parametrelere argüman denir. Verilen argümanları sırasıyla yazdırma işlemi default olarak adlandırılır.

Bir örnek üzerinde inceleyelim.

Örnek:

```
fun main(Args :Array<String>) {

bilgileriYazdir("Beyza","Koyulmuş",52)

}
fun yazdir(message:Any)
{
    println(message)
}
fun bilgileriYazdir(adi:String, soyadi:String, yasi:Int){
    yazdir(adi)
    yazdir(soyadi)
    yazdir(yasi)
}
```

```
Beyza
Koyulmuş
52

Process finished with exit code 0
```

Örnekte parametreler sırasıyla çağırılmıştır. Yani

`fun bilgileriYazdir(adi:String, soyadi:String, yasi:Int)` buradaki sıra gözönüne alınarak `bilgileriYazdir("Beyza","Koyulmuş",52)` şeklinde önce ad, sonra soyad daha sonrada yaş bilgisi girilmiştir. Fakat bazı durumlarda parametlereleri farklı sıra da girmemiz gereken durumlar olabilir.

Mamed Arguments

Verilen argümanları farklı sırasıyla yazdırma işlemi Mamed Arguments olarak adlandırılır.

Örnek:

```
fun main(Args :Array<String>) {
bilgileriYazdir(yasi=52,soyadi="Koyulmuş",adi="Beyza")

}
fun yazdir(message:Any)
{
    println(message)
}
fun bilgileriYazdir(adi:String, soyadi:String,yasi :Int){
    yazdir(adi)
    yazdir(soyadi)
    yazdir(yasi)
}
```

```
Beyza
Koyulmuş
52

Process finished with exit code 0
```

VarArg Kullanımı

- Eğer fonksiyona verilecek parametreler belli değilse, “vararg” kullanabiliriz.
- Aslında boyutu belli olmayan bir liste diyebiliriz.
- Bir fonksiyonda sadece 1 parametre vararg olabilir.
- yüzden argüman olarak farklı tipte değerler veya diziler verebiliriz.

Örnek:

```
fun main(Args :Array<String>) {

isimleriYazdir("Beyza","Hamza","Kübra","Ayşe")
}

fun isimleriYazdir(vararg isim :String)
{
    for(x in isim)
    {
        println(x)
    }
}
```

```
Beyza
Hamza
Kübra
Ayşe

Process finished with exit code 0
```

Extention (Genişletilmiş) Fonksiyonlar

Extension'lar bir sınıfı inherit etmeden yeni bir özellik kazandırmak amacıyla yapılır.

Örnek:

```
fun main(Args :Array<String>) {

   val sonuc=3.karesinial()

    print(sonuc)

}


fun Int.karesinial():Int{

    return this*this


}
```

```
9
Process finished with exit code 0
```

Örnek: Kullanıcının girdiği ismi Sayın “kullanıcı ismi” Hoşgeldiniz , şeklinde return eden programı yazınız.

```
fun main(Args :Array<String>) {
   println("İsminizi giriniz: ")

  val isim = readLine()
  print(isim?.karsila())

}

fun String.karsila():String{
    return "Sayın "+this+" Hoşgeldiniz...."
}
```

```
İsminizi giriniz:
Beyza KOYULMUŞ
Sayın Beyza KOYULMUŞ Hoşgeldiniz....
Process finished with exit code 0
```

Extention fonksiyonlar ile aynı zamanda bileşke fonksiyon olarak yani farklı fonksiyonları birlikte kullanabiliriz. Bir örnekte inceleyelim.

Örnek: 3 sayısının küpünü alan ve tekliğini kontrol eden ayrı extention fonksiyonlarını ve 3 sayısının küpünün tek olup olmadığını kontrol eden bileşke extention fonksiyonunu yazalım.

```
fun main(Args :Array<String>) {

    println("3 sayısının küpü "+3.kupAl())
    println("3 sayısı tek mi : "+3.tekMi())

    println("3 sayısının küpü tek mi : "+  3.kupuTekMi())

}

    fun Int.kupAl():Int{
    return this*this*this
    }

    fun Int.tekMi():Boolean{
        return this%2==1
    }


    fun Int.kupuTekMi():Boolean{
        return kupAl().tekMi()
    }
```

```
3 sayısının küpü 27
3 sayısı tek mi : true
3 sayısının küpü tek mi : true

Process finished with exit code 0
```

Infix Fonksiyonlar

- Daha okunabilir bir kod yazılımı sağlar. Fonksiyonun başında infix anahtar kelimesi kullanılır.
- Member function (üye fonksiyon/bir sınıfa ait fonksiyon) veya extension fonksiyon olmalıdır.
- Infix fonksiyonlar 1 parametre almak zorundadır. Daha fazla veya daha az parametre alamaz.
- Infix fonksiyonlar default değer kabul etmezler ve vararg değişken kabul etmezler.

Örnek:

```
fun main(Args :Array<String>) {

    var isim="Beyza"
    print(isim ekle "Koyulmuş")

}
private infix fun String.ekle(soyisim: String ) : String{
    return " " + this +" " +soyisim
}
```

```
Beyza Koyulmuş
Process finished with exit code 0
```

Fonksiyonların Scop'u

- Sınıfın içinde yazılan fonksiyonlara member function (üye fonksiyon)denir.
- Eğer bir fonksiyon, sınıf içinde değilde bir dosya içinde tanımlanıyorsa, buna "top level function" denir.
- Kotlin'de bir fonksiyon içinde başka bir fonksiyon tanımlanabilir. Buna local fonksiyon denir.
- Local fonksiyonlar kodların yeniden kullanmasını istendiğinde ancak member (üye) fonksiyon yapılmak istenmediği durumda kullanılabilir. Çünkü yalnızca bir fonksiyondan çağrılacaktırç
- Local fonksiyonlar classlardaki member fonksiyon sayısının azaltılmasına ve kodu yeniden kullanmaya yardımcı olur.
- Local fonksiyonlara sadece bulunduğu fonksiyon içinden erişilebilir.

Örnek:

```
class Hafta3{
    fun uyeFonksiyon(){

       println("Bu bir üye fonksiyondur")

        fun lokalFonksiyon(){

            println("Bu bir lokal fomksiyondur")
        }

    lokalFonksiyon()

    }

}


fun topLevelFonksiyon()
{

    println("Bu bir topLevelFonksiyondur")

}



fun main(Args :Array<String>)
{

   topLevelFonksiyon()

   var hafta3=Hafta3()

   hafta3.uyeFonksiyon()

}
```

```
Bu bir topLevelFonksiyondur
Bu bir üye fonksiyondur
Bu bir lokal fomksiyondur

Process finished with exit code 0
```

Aşırı Yüklenme (Method Overloading)

Bir fonksiyonun aynı isim ile birden fazla yazılmasıyla oluşturulur. Aynı isimle yazılan fonksiyonların parametreleri farklı olmalıdır. Örneğin bir fonksiyonda tipi integer olurken diğer fonksiyonda string tipinde parametre girilmesiye oluşturulabilir. Veya girdi sayıları farklı farklı olabilir. Örneğin bir fonksiyon bir parametre alırken, diğer fonksiyon iki parametre alır. Aynı sayıda parametreye sahip fonksiyonlar için tipleri farklı olmalıdır.

Ana metodda fonksiyonun aldığı değere göre karar verip ilgili fonksiyon çalışır.

Örnek:

```
fun yazdir(){
   println("Merhaba")
}

fun yazdir(mesaj: String)
{
   println(mesaj)
}

fun yazdir(sayi:Int)
{
   println(sayi)
}

fun yazdir(mesaj1: String, mesaj2:String )
{
   println(mesaj1)
   println(mesaj2)
}

fun main(Args :Array<String>)
{
   yazdir("Beyza")
   yazdir (4)
   yazdir()
   yazdir("Beyza","Koyulmuş")

}
```

```
Beyza
4
Merhaba
Beyza
Koyulmuş

Process finished with exit code 0
```

Örnek: Bir sayının asal olup olmadığını bulan programı fonksiyon kullanarak yazalım.

```
fun main(Args :Array<String>) {

asalMi(7)

}

fun asalMi(sayi:Int){
    var adet=tamBolenAdetBul(sayi)
    if(adet==2){
        println("Sayi Asaldir")
    }
    else
    {
        println("Sayi Asal değildir.")
    }
}


fun tamBolenAdetBul(sayi:Int):Int{

    var adet=0
    for(x in 1..sayi)
    {
        if(sayi%x==0){
            adet++
            println(""+sayi+" sayisi "+x+"'e bölünür ")
        }
        else
        {
            println(""+sayi+" sayisi "+x+"'e bölünmez ")
        }
    }
    return adet
}
```

```
7 sayisi 1'e bölünür
7 sayisi 2'e bölünmez
7 sayisi 3'e bölünmez
7 sayisi 4'e bölünmez
7 sayisi 5'e bölünmez
7 sayisi 6'e bölünmez
7 sayisi 7'e bölünür
Sayi Asaldir

Process finished with exit code 0
```

Kotlin'de Nesne Tabanlı Programlamaya Giriş

Nesne Tabanlı Yaklaşım Nedir?

Nesne Tabanlı Yaklaşım, **yeniden kullanılabilirliğe** odaklanan çözümler tasarlama ve uygulama felsefesi ve yoludur. Bu yaklaşım temel bileşenlerin icadı olan sanayi devriminden esinlenmiştir. Örneğin evimizi yaparken kendi tuğlalarımızı ve çivileri yapmıyoruz. Sipariş veriyoruz.

Bir "Bileşen Programı" yapmak daha akıllı ve daha ucuzdur. Bileşenler bozulmaz, test edilir ve bakımı yapılır. Bir sorun varsa, bu büyük olasılıkla yazdığımız kodda belirli bir konumdadır. Yazdığımız kodlar başkaları tarafından veya başka projelerde kendimiz tarafından kullanılabilir.

Elbette şimdiye kadar edindiğimiz bilgileri de kullanacağız. Temel fark, şimdi kodumuzun birden çok iletişim nesnesine farklı şekilde yapılandırılmasıdır.

Nesne Tabanlı Programlama (Object Orianted Programming(OPP))

Yazılım tasarımını; **işlevler ve mantık yerine, veri veya nesneler etrafında düzenleyen** bir programlama dili modelidir. Günümüzde yazılım geliştirme teknolojileri içerisinde büyük bir önem taşır. Kod karmaşıklığını ortadan kaldıran, daha performanslı ,daha rahat yazılabilen , okunabilen ve daha geliştirilebilir kod yazmayı hedefler.

Nesne tabanlı programlamada her işlev soyutlanır, izole edilir. Yani yapacağımız her işlev bir birinden ayrılır, birbirlerine direkt bağlantıları yoktur.

Nesne Tabanlı Programlama Faydaları

- Nesne oluşturma bir sınıf içerisinde toplanır ve tüm projelerde **kullanılabilirliğe** olanak sağlar.
- Sınıfların bir kez oluşturulması sayesinde uzun kodları **tekrardan yazmak yerine kısa kodlamalar** ile çalıştırılabilir.
- Uzun kodların tekrar yazılmasının engellenmesi sayesinde **geliştirme süreci kısalır**.
- Nesneler birbirinden bağımsız olduğundan **bilgi gizliliği** konusunda avantaj sağlar.
- Sınıflar sayesinde tüm projelerde değişiklik yapmak yerine tek bir sınıfta değişiklik yapılıp tüm projelerde çalışması sağlanır. Bu zaman kaybını büyük ölçüde azaltır.

Nesne tabanlı programlamanın **temelinde 2 şey mevcut**tur:

1. **Object (Nesne)**
2. **Class (Sınıf)**

- Her şey **Object**dir (Nesne) ve her nesne bir **Class'a**(Sınıf) aittir.

Bu, günlük hayatımızda da böyledir. Etrafımızdaki her şey bir nesnedir. Bu nesneler muhakkak bir sınıfa aittir. Örneğin canlılar sınıfına baktığımızda içerisinde hayvanlar , bitkiler sınıfı yer alır. Bitkiler sınıfı içerisinde meyveler ve sebzeler sınıfı mevcut. Meyveler sınıfında elma, çilek, muz gibi nesneler vardır.

Object (Nesne) Nedir ?

Nesnelerin 2 tane önemli özelliği vardır.

- **Durum** (Attribute, Properties) ve **Davranış** (Behaviour)

  Bunu bir örnek üzerinde açıklayalım. Araba nesnesini ele alalım. Arabanın rengi, fiyatı, modeli gibi özellikler arabanın **durumu**dur. Arabanın çalıştırılabilmesi, hızlanabilmesi ve yavaşlayabilmesi gibi özellikleri de arabanın **davranış**larıdır.
  Arabayı nesne olarak düşündüğümüzde durumu (model, renk, fiyat gibi özellikler) **değişkenler** üzerinde, davranışları (çalıştırma, park etme gibi özellikler) da **metotlarda** tutulur.

Class (Sınıf)

Sınıflar, herhangi bir nesne yönelimli programlama dilinin ana yapı taşlarıdır. Tüm nesneler bir sınıfın parçasıdır ve sınıf tarafından veri üyeleri ve üye işlevler biçiminde tanımlanan ortak özelliği ve davranışı paylaşır.

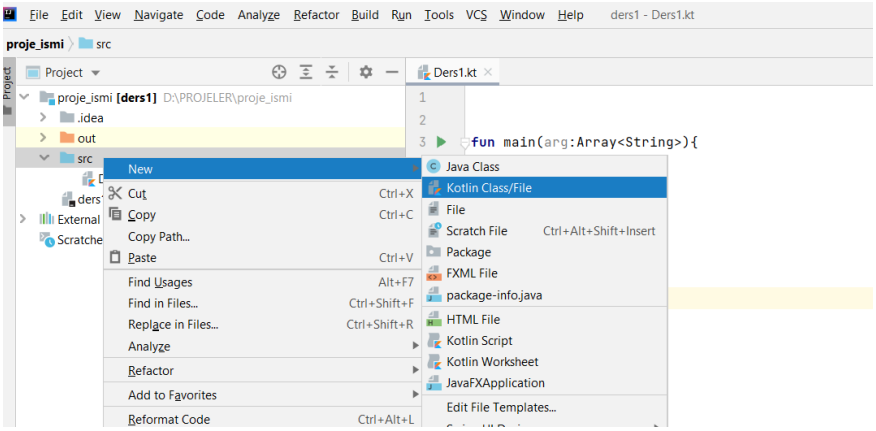
Bir nevi sınıflar, nesnelerin durum ve davranışlarını tutan yapılardır. Yani aslında değişkenleri ve metotları bir arada saklayan sistemdir.

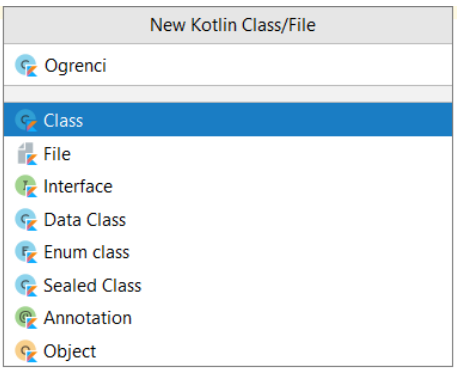
- Değişkenler kullanacağımız verileri örneğin ad, soyad ve yaş gibi verileri saklar.
- Metotlar bu verilerin toplanması gibi görevleri sağlayan bir sistemdir.

Sınıf Oluşturmak

Kotlin'de sınıf oluşturmak için class anahtar sözcüğü kullanılır veya aşağıda gösterildiği şekilde class oluşturulur.

Src > sağ tık > New > Kotlin Class/File





- Açılan sayfada Class'ı seçip , Class ismini yazıyoruz.
- Ogrenci adında bir class açtık.

Örnek: Ogrenci adındaki class içerisine öğrencilerin not ortalamalarını hesaplayan bir fonksiyon yazalım.

Ogrenci.kt

```
class Ogrenci {

    var vize:Double?=null
    var final:Double?=null

    fun getOrtalama():Double
    {
        return(this.final!!*0.6)+(this.vize!!*0.4)
    }
}
```

Ders1.kt

```
fun main(arg:Array<String>){

 var ogrenci=Ogrenci()

 ogrenci.vize=60.0

 ogrenci.final=90.0


 println("Ogrenci ortalaması : "+ogrenci.getOrtalama())

}
```

```
Ogrenci ortalaması : 78.0

Process finished with exit code 0
```

- Örnekte görüldüğü gibi sınıftakilere erişmek için, bir nesne tanımlaması yapmamız gerekiyor. Örneğin Ogrenci sınıfına ulaşmak için ogrenci adında bir nesneyi şu şekilde oluşturduk: **var** ogrenci=Ogrenci()

Object (Nesne)

Nesneler, sınıfın özellikleri ve davranışlarını kullanır. Sınıfa herhangi bir bellek tahsis edilmez. Sınıfın nesneleri oluşturulduktan sonra bellekte yer kaplarlar. Sınıfın üyelerini ve üye işlevlerini kullanarak veriler üzerinde çeşitli eylemler gerçekleştirirler.

Aslında verileri saklayan ve bu veriler üzerinde işlem yapan metotları saklayan bileşenlerdir, denilebilir.

Örneğin, **Ogrenci adlı bir sınıf**a erişmek için nesne tanımlaması aşağıdaki gibi yapılmaktadır :

Nesne tanımlaması:	**Örnek:**

`var nesneAdi= erişilecekSinif()`
 	
`veya`
 	
`var nesneAdi= erişilecekSinif()`
	`fun main(arg:Array<String>){`
	
	`var sinifNesnesi= Ogrenci()`
	
	`}`

Not: Burada nesne olarak tanımladığımız **sinifNesnesi** adlandırılmasında dikkat etmemiz gereken, Kotlin'de önceden tanımlanmış **anahtar kelimeler dışında** nesneye herhangi bir isim verebiliriz. Bir başka dikkat edilmesi gereken nokta da nesneyi oluşturmak için new anahtar sözcüğünü kullandığımız Java'dan farklı olarak biz burada new anahtar sözcüğünü kullanmıyoruz, eğer kullanılırsa derleme hatası alırız.

Örnek: Beş sayısının karesini bulma işlemini Hesapla adında bir class açıp içerisinde karesiniHesapla adında bir fonksiyon ile bulalım.

Hesapla.kt

```
class Hesapla {
    var sayi: Int = 5
    fun karesiniHesapla(): Int {

        return sayi*sayi
    }
}
```

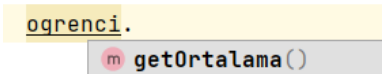
Ders1.kt

```
fun main(arg:Array<String>)
{
    val obje=Hesapla()

    // Hesapla() class'ının obje adında bir nesnesini oluşturduk

    println ( "Beş sayısının karesi: "+obje.karesiniHesapla())
    /*obje isimli nesne sayesinde Hesapla() classındaki
     karesiniHesapla() fonksiyonuna ulaştık.*/
}
```

```
Beş sayısının karesi: 25

Process finished with exit code 0
```

Nokta ve This

- Nesnenin sınıf içerisindeki özelliklerine erişebilmek için nokta(.) operatörünü kullanmamız gerekiyor. Az önce yaptığımızdan örnek verecek olursak :



şeklinde kullanılıyor.

- This ise OOP'de mevcut nesneye atıfta bulunmak için kullanılır. Aynı class içerisindeki scop'u ifade ediyor.
  Az önceki örnek üzerinde bakacak olursak : `this.final` şeklinde kullanmıştık. Aslında burada mevcut class içerisindeki `final` değişkeni ile işlem yaptığımızı belirtiyoruz.

Örnek:

Ogrenci.kt

```
class Ogrenci {

    var adi:String?=null
    var no:Int?=null
    var yasi:Int?=null

    fun bilgileriGoster()
    {
        println("Öğrenci adı :" +this.adi)
        println("Öğrenci yaşı :" + this.yasi)
        println("Öğrenci no :" + this.no)
    }
}
```

Ders1.kt

```
fun main(arg:Array<String>) {

 var nesne = Ogrenci()

 nesne.adi="Beyza"
 nesne.yasi=37
 nesne.no=1272320

 nesne.bilgileriGoster()

}
```

```
Öğrenci adı :Beyza
Öğrenci yaşı :37
Öğrenci no :1272320

Process finished with exit code 0
```

Constructor (Kurucu) Metot

Kurucu metot olarak da bilinen Constructor temel amacı, bir sınıfın özelliklerini başlatmaktır. Bir sınıfın nesnesini oluşturduğumuzda constructor çağrılır.

- Nesne tabanlı programlama dillerinde kurucu ya da yapıcı diye çevirdiğimiz bileşenler,nesne oluştuğu andan itibaren bazı kodların ya da bazı atamaların yapılmasını sağlar.
- Kurucular genellikle oluşturulan nesnelere veri sağlamak için kullanılır.
- Java'da kurucular sınıf içerisinde tanımlanan ve sınıf adıyla aynı ismi taşıyan metotlar iken, Kotlin'de ise durum farklıdır.
- Kurucu metot yerine çalıştırılacak kod bloğu bulunur.

Constructor, bir sınıf oluştururken değerlerinin nasıl olacağını veriyor. Parametre alacaksa parametrelerini veriliyor.

İki şekilde constructor oluşturabiliriz:

1.yöntem:

- **constructor** anahtar kelimesiyle aşağıdaki örnekteki gibi oluşturabiliriz.
- Az önceki örnekte nesneyi boş olarak oluşturuyorduk fakat şuan nesne oluşturulurken değer vereceğiz.

SinifAdi.kt

```
class SinifAdi constructor(parametre1:türü, parametre2:türü ,…) {

var değişken1:türü= parametre1

var değişken2:türü= parametre2

}
```

Main.kt

```
fun main(arg:Array<String>) {


var nesne = SinifAdi(

            parametre1 değeri,

            parametre2 değeri,
                ...
            )
}
```

Artık burada nesneyi oluşturunca içerisine değerler verebiliyoruz.

Örnek :

Ogrenci.kt

```
class Ogrenci
constructor(adiParametre:String,noParametre:Int,yasiParametre:Int) {
    var adi:String?=adiParametre
    var no:Int?=noParametre
    var yasi:Int?=yasiParametre

    fun bilgileriGoster()
    {
        println("Öğrenci adı :" +this.adi)
        println("Öğrenci yaşı :" + this.yasi)
        println("Öğrenci no :" + this.no)
    }
}
```

Ders1.kt

```
    fun main(arg:Array<String>) {

    var nesne = Ogrenci(

                "Beyza", //adiParametre

                1237258, //noParametre

                37       //yasiParametre

            )
    nesne.bilgileriGoster()
}
```

```
Öğrenci adı :Beyza
Öğrenci yaşı :37
Öğrenci no :1237258

Process finished with exit code 0
```

2.yöntem,init ile oluşturma şu şekilde yapılabilir :

SinifAdi.kt

```
class Ogrenci
 {

   init{

        // başlangıçta çalışmasını istediğimiz blok

    }
}

//programda ilk olarak init çalışır.
```

Main.kt

```
fun main(arg:Array<String>) {

 var nesne = SinifAdi()
             //Burayı oluşturduğumuz an da init çalışır.


}
```

Örnek :

Ogrenci.kt

```
class Ogrenci
 {

   init{
        println("init çalıştı")
    }
}

//programda ilk olarak init çalışır.
```

Ders1.kt

```
fun main(arg:Array<String>) {

 var nesne = Ogrenci()


}
```

```
init çalıştı

Process finished with exit code 0
```

Örnek :

Ogrenci.kt

```
class Ogrenci(var adi:String, var soyadi:String){

init{

println("Nesne oluşturuldu. Gönderilen değişkenler değişkene atandi.")


println("Adi: "+adi)


println("Soyadi: "+soyadi)

 }

}
```

Ders1.kt

```
fun main(arg:Array<String>) {

 var ogrenci = Ogrenci(
  "Beyza",
  "Koyulmuş"
 )

}
```

```
Nesne oluşturuldu. Gönderilen değişkenler değişkene atandi.
Adi: Beyza
Soyadi: Koyulmuş

Process finished with exit code 0
```

Nesne Tabanlı Programlamanın Özellikler

Sistemin bir çok özelliği vardır. Bunları 4 temel özellikte inceleyebiliriz.

- Kalıtım (Inheritance)
- Soyutlama (Abstraction)
- Kapsülleme (Encapsulation)
- Çok Biçimlilik (Polymorphism)

Kalıtım (Inheritance)

Kalıtım, bir sınıfın başka bir sınıfın tüm özelliklerini miras aldığı bir özelliktir. Özelliklerin miras alındığı sınıf; temel sınıf, süper sınıf veya üst sınıf olarak bilinir ve özellikleri miras alan sınıf ise; türetilmiş sınıf veya alt sınıf olarak bilinir.

Neden Kalıtım Kullanılır?

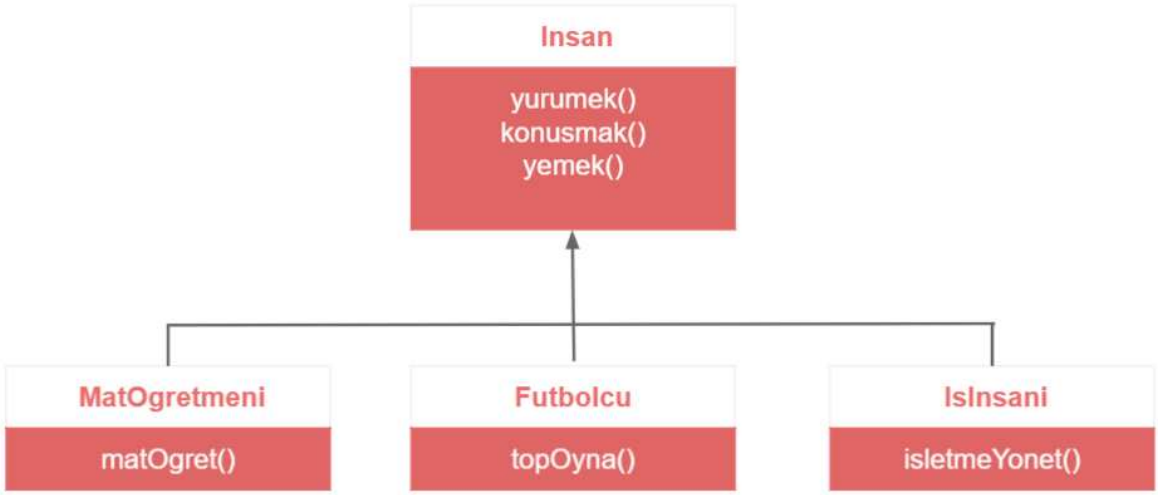
Diyelim ki uygulamamızda üç karakter istiyoruz; bir matematik öğretmeni, bir futbolcu ve bir iş insanı olsun.

Karakterlerin hepsi insan olduğu için yürüyebilir ve konuşabilirler. Bununla birlikte bazı özel yetenekleri de vardır. Bir matematik öğretmeni matematik öğretebilir, bir futbolcu çok iyi top oynayabilir ve bir iş insanı işletme yönetebilir.

Yürüyebilen, konuşabilen ve özel becerilerini sergileyebilen üç sınıf oluşturabiliriz. Bunu tablo üzerinde göstermemiz daha iyi olacaktır :

Sınıfların her birinde, her karakterde yürümek ve konuşmak fonksiyonları için aynı kodu kopyalayacaktık. Yeni bir özellik eklemek istediğimizde yemek fonksiyonu eklediğimizi düşünelim, her karakter için aynı kodu uygulamamız gerekir. Bu, kodu kopyalarken kolayca hataya açık hala gelebilir ve kod satır sayımız çoğalır.

Insan adında; konuşma, yürüme, yemek yeme, uyuma gibi temel özelliklere sahip bir sınıfımız olsaydı ve sadece oluşturduğumuz karakterimize göre özel becerileri ekleseydik çok daha kolay olurdu. İşte bu işlem kalıtım kullanarak yapılır.



Kalıtım kullanarak, artık her sınıf için yurumek(), konusmak() ve yemek için aynı kodları tekrar yazmamıza gerek yok. Sadece artık onları miras almamız gerekiyor.

Böylece MatOgretmeni (türetilmiş sınıf) için Insan sınıfının tüm özelliklerini devralırız ve sadece MatOgretmeni sınıfına ait olan özelliği buraya yazarız. Benzer şekilde Futbolcu sınıfı için, Insan sınıfının tüm tüm özelliklerini devralır ve topOyna() gibi yeni bir özelliği bu sınıfa ekleriz. Kalıtım, kodu daha temiz ve daha anlaşılır yapmaktadır.

Özetle kalıtım özelliklerini maddeler halinde yazacak olursak :

- Alt ve üst sınıfın ortak özelliklerinden alabilme sistemine Miras Alma denir.
- Gerçek hayatta bulunan kalıtım yapısından esinlenerek kullanılan Inheritance, özelliklerini kalıtım vasıtasıyla başka nesnelere aktarabilir.
- Android programlamada kullanılan Object sınıfı tüm sınıfların atasıdır. Diğer sınıflar bu Object sınıfından türetilmiştir.
- Kotlin'de ise tüm sınıflar Any sınıfından türetilmiştir.
- Kalıtım kullanarak kod tekrarından kurtulabiliriz. Daha kısa ve clean code yazmış oluruz.
- Kotlinde kalıtım **: (iki nokta üst üste)** ile yapılır ve class erişimi **open** olur. Bir sınıfın miras verebilmesi için o sınıfa **open** belirteci eklenmelidir. Kotlinde sınıflar default olarak miras alınamaz yapılardır.

Örnek:

Insan.kt

```
open class Insan(isim :String, yas:Int) {

  //open anahtar kelimesi ile bu sınıfın kullanılmasına izin veriliyor.

    init{

        println("Benim adım :$isim")

        println("yaşım : $yas")
    }
}
```

IsInsani.kt

```
class IsInsani (isim:String,yas:Int):Insan(isim,yas) {

    fun isletmeYonet(){

        println("Ben kendi şirketimi yönetiyorum")
    }
}
```

MatOgretmeni.kt

```
class MatOgretmeni(isim:String,yas:Int) : Insan (isim,yas) {

    fun matOgret(){

        println( "Ben ilkokul öğrencilerine matematik öğretiyorum")
    }
}
```

Ders1.kt

```
fun main(args: Array<String>) {

    val  mat = MatOgretmeni("Zehra", 25)
    mat.matOgret()

    println()

    val obje=IsInsani("Hamza",32)
    obje.isletmeYonet()

}
```

```
Benim adım :Zehra
yaşım : 25
Ben ilkokul öğrencilerine matematik öğretiyorum

Benim adım :Hamza
yaşım : 32
Ben kendi şirketimi yönetiyorum

Process finished with exit code 0
```

Örnek:

Araba.kt

```
open class Araba {

//open anahtar kelimesi ile bu sınıftan bir sınıf türetilmesine izin
vermiş oluyoruz.

    var marka:String?=null

    var renk:String?=null

    var yil:Int?=null



    open fun calistir(){

        println("Benzinli  motor aktif")
    }


    override fun toString():String{

        return "Özellikler(marka=$marka, renk=$renk, yıl=$yil)"
    }

}
```

KirmiziAraba.kt

```
class KirmiziAraba : Araba(){

 init {
        renk="Kirmizi"
        yil=2000
        marka="Mercedes"
    }
}
```

Ders1.kt

```
fun main(arg:Array<String>) {

 var araba=Araba()
 var kirmiziAraba=KirmiziAraba()

 println(araba.toString())
 println(kirmiziAraba.toString())

 araba.calistir()

 kirmiziAraba.calistir()


}
```

```
Özellikler(marka=null, renk=null, yıl=null)
Özellikler(marka=Mercedes, renk=Kirmizi, yıl=2000)
Benzinli  motor aktif
Benzinli  motor aktif

Process finished with exit code 0
```

Override ve Super İfadesi

Üst sınıftaki fonksiyon ve özelliklere erişmek için override anahtarı kullanılır. Override aslında ezmek anlamı taşır yani gerektiği durumlarda üst sınıftan aldığımız mirası kendi sınıfımıza özel değiştirebiliriz. Super ifadesi ile de bir üst sınıftaki metoda erişmek için kullanılır.

Örnek: Daha iyi anlaşılması için az önceki örneğimize bir elektrikli araba ekleyelim ve kalıtım aldığımız Araba sınıfındaki bir fonksiyonu kullanalım.

ElektrikliAraba.kt

```
class ElektrikliAraba: Araba() {

     override fun calistir(){
         println("Elektikli Araba Çalıştırıldı")

         println("Elektrikli motor aktif\n")

         super.calistir()
     }
}
```

KirmiziAraba.kt

```
class KirmiziAraba : Araba(){

  init {
        renk="Kirmizi"
        yil=2000
        marka="Mercedes"
   }
}
```

Araba.kt

```
open class Araba {
//open anahtar kelimesi ile bu sınıftan bir sınıf türetilmesine izin
vermiş oluyoruz.

    var marka:String?=null
    var renk:String?=null
    var yil:Int?=null

    open fun calistir(){

        println("Araba Çalıştırıldı")

        println("Benzinli motor aktif\n")

    }
    override fun toString():String{

        return "Özellikler(marka=$marka, renk=$renk, yıl=$yil)"
    }
}
```

Ders1.kt

```
fun main(arg:Array<String>) {

 var araba=Araba()
 var kirmiziAraba=KirmiziAraba()

 var elektrikliAraba=ElektrikliAraba()

 println(araba.toString())
 println(kirmiziAraba.toString())

 araba.calistir()
 kirmiziAraba.calistir()
 elektrikliAraba.calistir()
}
```

```
Özellikler(marka=null, renk=null, yıl=null)
Özellikler(marka=Mercedes, renk=Kirmizi, yıl=2000)
Araba Çalıştırıldı
Benzinli motor aktif

Araba Çalıştırıldı
Benzinli motor aktif

Elektikli Araba Çalıştırıldı
Elektrikli motor aktif

Araba Çalıştırıldı
Benzinli motor aktif
```

Kapsülleme (Encapsulation)

Kapsülleme, nesnenin özellikleri üzerinde okuma ve yazma iznini kontrol eden mekanızmadır. Veri Gizleme (data hiding) olarak da bilinir.

- Kapsülleme bir nesnenin iç yapısını (verilerini ve özelliklerini) dış dünyadan doğrudan erişime kapatılması anlamına gelir.
- Bu sayede nesneye ait veriler değer ataması yapılırken yanlış kullanımdan **korunmuş** olunur. Bu koruma sayesinde yanlış kullanımlar için önlem alınmış olur.
- İç yapısının dışarıya açık olmaması aynı zamanda iç yapı ile ilgili değişikliklerin dış dünyanın etkilenmemesi de sağlanmış olur.

Erişim Belirleyiciler(Visibility) / (Access Modifier)

Kotlin'de 4 adet erişim belirleyici vardır:

- **public :** Her yerden erişilebilir. Public kelimesi halka açık anlamına gelmektedir. Bu erişim belirleyicisi ile tanımlanan her şey diğer bütün sınıflardan erişilebilir. (Java ile aynı)
- **private :** Sadece tanımlandığı sınıftan erişilebilir. Private özel veya gizli anlamına gelir. Bu erişim belirleyicisi ile tanımlanan herhangi bir eleman sadece ve sadece aynı sınıf içerinden erişilebilir diğer sınıflardan erişemez. (Java ile aynı)
- **protected** : Aynı sınıftan ya da o sınıftan türetilen sınıflar, nesneler tarafından erişilebilir.
- **internal :** Aynı modül içindeki diğer sınıflar erişebilir. (Java'da yok)

Not: Eğer herhangi bir erişim belirleyici tanımlamazsak **default olarak public** olacaktır.

Örnek: Erişim belirleyicilerin her bir türünden birer fonksiyon yazıp erişilme durumlarını inceleyelim.

Test.kt

```
class Test {

    public fun publicFonksiyon(){
        print("Public Fonksiyon ")
    }

    private fun privateFonksiyon(){
        print("Private Fonksiyon ")
    }

    protected fun protectedFonksiyon(){
        print("Protected Fonksiyon ")
    }

    internal fun internalFonksiyon(){
        print("Internal Fonksiyon ")
    }

}
```

Ders1.kt

```
fun main(arg:Array<String>) {

 var testObje=Test()


 testObje.publicFonksiyon()
 // public e erişim var.


 //testObje.privateFonksiyon()
 // Hata (Erişim yok)


 //testObje.protectedFonksiyon()
 // Hata (Erişim yok)



 testObje.internalFonksiyon()
 //Erişim var

 }
```

```
fun main(arg:Array<String>) {

 var testObje=Test()

 testObje.publicFonksiyon()
 // public e erişim var.

 testObje.privateFonksiyon()
 // Hata (Erişim yok)

 testObje.protectedFonksiyon()
 // Hata (Erişim yok)

 testObje.internalFonksiyon()
 //Erişim var
}
```

```
Public Fonksiyon Internal Fonksiyon
Process finished with exit code 0
```

Getter ve Setter

Genellikle bir nesne özelliğinin, sınıfın dışından nasıl değiştirildiğini kontrol etmek isteriz. Özelliği salt okunur olarak ayarlamak veya değişikliklerine bir şekilde tepki vermek isteriz. Sınıf değişkenlerini okumak ve yazmak için getter ve setter kullanıyoruz. Kotlin' de getter ve setter tanımlamak zorunda değiliz, biz yazmasakta arka planda tanımlanıyor.

Fakat bu kavramları daha iyi anlamak ve neden gerekli olduklarını bilmek için örnek üzerinde anlatalım:

```
class User(isim:String, yas:Int,email:String) {


    var isim : String =isim

    var yas: Int = yas

    var email: String = email

}

 fun main(){

    val user = User("Kübra", 20, "kubra@gmail.com")
    user.yas = -10

}
```

- Yukarıda User adında bir class tanımladık. Ana fonksiyonun içinde yas değerini negatif bir değer ayarladık. (-10)
- Fakat herhangi bir kullanıcının yaşını negatif olarak girmesini istemeyiz.
- Böyle durumları önlemek için istediğimiz şartı setter içinde yazabiliriz. Örneğin yas özelliği için bir ayarlayıcının nasıl tanımlanacağına bakalım:

```
class User(isim:String, yas:Int,email:String) {

    var isim : String =isim
    var yas: Int = yas

        set(value) {
            if(value < 0)
            {
                println("Yaş negatif olamaz")
            }
            else
            {
                field = value
            }
        }

    var email: String = email
}
```

```
Yaş negatif olamaz

Process finished with exit code 0
```

- Yukarıdaki kodda yas özelliği üzerinde bir setter oluşturduk. Değerin negatif olmaması için bir if şartı getirdik.

- **set**(value) içindeki value sadece bir değişken adıdır, farklı bir adlandırma da yapılabilir. Bu, özelliğe atanan değeri tutmaktadır.

- **Set** içine, atanan değerin negatif olup olmadığını kontrol etmek için **if- else** koşulu ekledik. Değer negatifse " Yaş negatif olamaz " yazdıracak.

- Diğer durumda, değeri fielde atadık.(Özelliği işaret eden özel bir değişken). Field, değeri o özelliğe ayarlamak için setter içinde kullanılır. (field=value)

Getter

- Tıpkı setter(ayarlayıcılar) gibi, getter(alıcıları) da tanımlayabiliriz. Özelliğe erişirken, alıcının içindeki kod yürütülür.
- Bu alıcı işlevleri, değere erişirken herhangi bir özellik değerini biçimlendirmek için kullanılır. Bunu bir örnek üzerinde görelim:

```
class User(isim:String, yas:Int,email:String) {

    var isim : String =isim
    var yas: Int = yas
    var email: String = email
        get(){
            return field.toLowerCase()
            //String değerindekini küçük harflere çevirir.

              }
      }
fun main(){

    val user = User("Kübra", 20, "KUBRA@gmail.com")
    println(user.email)

}
```

```
kubra@gmail.com

Process finished with exit code 0
```

- Yukarıdaki kod parçacığında, e-posta özelliğine eriştiğinizde yürütülen bir alıcı işlevi tanımladık.
- Kullanıcı ne girmiş olursa olsun, e-postalar genellikle küçük harfla yazılır. Her zaman küçük harfli olarak biçimlendiririz.
- Bunu yapmak için, e-mail özelliğini küçük harf formatında döndürecek bir alıcı tanımladık. E-mail de "KUBRA@gmail.com" girilmesine rağmen çıktı "kubra@gmail.com" yazdırıldı.

Ogrenci adında bir class açalım.

Ogrenci.kt

```
class Ogrenci(var isim: String, var kadin: Boolean, var yas: Int) {

    var resit: Boolean

    init {
        resit = true
        if (yas < 18)
            resit = false
    }

    override fun toString(): String {
        var resitMi = "reşit"
        if (!resit)
            resitMi = "reşit değil"

        var cinsiyet ="kadın"
        if (!kadin)
            cinsiyet = "erkek"

        return "Adı : $isim, $cinsiyet. Yaşı : $yas ve $resitMi"
    }
}
```

- Oluşturduğumuz sınıfta öğrencinin adı, cinsiyeti ve yaşı var. resitMi değişkenimizi yaşa göre ayarlandık. 18 ve üzeri yaşındaysa reşit, değilse reşit değildir. Cinsiyet verisinin türünü Boolean yaptık ve true değerini kadın, false değerini erkek olarak belirledik.

- Yapıcıyı kullanarak bir öğrenci oluşturalım:

Main.kt

```
fun main(arg:Array<String>){

  val ogrenci = Ogrenci( "Mert CAN" , false, 25 )
  println(ogrenci)

}
```

```
Adı : Mert CAN, erkek. Yaşı : 25 ve reşit

Process finished with exit code 0
```

- Buraya kadar her şey güzel görünüyor. Ancak özellikler yeniden yazılabilir, değerlere dışarıdan yaptığımız bir değişiklik ile düzgün çalışmayabilir. (Yani tutarsız bir duruma sahip olabilir. **Ogrenci sınıfı aynı,** Main sınıfını şu şekilde yazarsak:

Main.kt

```
fun main(arg:Array<String>){

  val ogrenci = Ogrenci( "Mert CAN" , false, 25 )
  ogrenci.yas=17
  ogrenci.kadin=true
  println(ogrenci)

}
```

```
Adı : Mert CAN, kadın. Yaşı : 17 ve reşit

Process finished with exit code 0
```

- Nesne sayesinde **yas** ve **cinsiyet** değerlerini değiştirdik. Kesinlikle yas değiştiğinde **resitMi** durumunun güncellenmesini isteriz. Bunun dışında, örneğin bir cinsiyetin değiştirilmesine izin verilme gereği yoktur. Ancak, özellikleri okumak için erişilebilir tutmak istiyoruz, bu yüzden onları private yapmadık.

- Belirli özellikleri okuyabilmek için get yöntemleri oluşturacağız ve bu özellikleri dışarıdan değiştirilmelerini önlemek için private yapacağız.

Şimdi sınıfımız şu şekilde olacak:

Ogrenci.kt

```
class Ogrenci(private var isim: String, private var kadin: Boolean,

private var yas: Int) {

var resit:Boolean

    init {
        resit = true
        if (yas < 18)
            resit = false
    }

    fun getIsim(): String {
        return isim
    }

    fun getresit(): Boolean {
       return resit
    }

    fun getYas(): Int {
        return yas
    }

    fun Cinsiyet(): Boolean {
        return true
    }
```

```
    fun setYas(deger: Int) {
        yas = deger


        resit= true

        if (yas < 18) {

            resit=false
        }
    }

    override fun toString(): String {

        var resitMi = "reşit"

        if (!resit)
            resitMi = "reşit değil"

        var cinsiyet ="kadın"
        if (!kadin)
            cinsiyet = "erkek"

        return "Adı : $isim, $cinsiyet. Yaşı : $yas ve $resitMi"
    }
}
```

Main.kt

```
fun main(arg:Array<String>){


  val ogrenci = Ogrenci( "Mert CAN" , false, 12 )

    println(ogrenci)



}
```

```
Adı : Mert CAN, erkek. Yaşı : 12 ve reşit değil

Process finished with exit code 0
```

- Sadece bir değer döndüren yöntemler çok basittir. Yaşa göre Resit özelliğini gözden geçirmemiz daha mantıklı olacaktır. Private yaparak değişkenleri istediğimiz şekilde kafamıza göre değiştirme yapamayacağımızdan emin olduk. Artık tüm değişiklikleri kontrol edebilir ve gerektiğinde onlarla çalışabiliriz. Böylece sıkıntı çıkarabilecek tüm istenmeyen değişiklikleri önleyebiliriz.

- Değer döndürme yöntemlerine Getter ve değeri ayarlama yöntemlerine Setter denir.

- Constructor gibi diğer özellikleri düzenlemek için editOgrenci() metodu ekleyebiliriz. Öğrencinin adı, yaşı ve diğer özellikleri bu yöntem kullanılarak güncellenecektir. Ayrıca belirli değerleri değiştirmeye yönelik tüm girişimleri tek bir yerde halledebileceğimiz için, orada ayarlanan değerleri de doğruyabiliriz. Ancak methotları kullanarak özellikleri sormak zaman alır ve kafa karıştırıcı olabilir. Bu nedenle Kotlin bize başka sözdizimi de sağlar.

Private Set

Sadece özelliklerin dışarıdan ayarlanmasını engellemek istiyorsak private set erişim değiştiricisini kullanabiliriz. Getterlere (metotlara) ihtiyaç duymadan mükemmel kapsüllemeyi sağlayacak olan sınıfımızdaki özelliklerin önüne yerleştiririz.

Not : Kotlin, kurucunun kısaltılmış versiyonunda bu değiştiriciyi kullanmayı desteklemiyor. Bu yüzden özellikleri sınıfın gövdesinde tanımlayacağız. Artık get ile yazdığımız kısımları ve değişkenlerin private kısımlarını silebiliriz.

Classımız şu şekilde olacak :

```
class Ogrenci( isim: String,  kadin: Boolean, yas: Int) {



    var isim= isim
        private set

    var kadin = kadin
        private set

    var yas = yas
        private set

    var resit: Boolean=false
        private set
...

// uygulamanın geri kalanı
```

- Artık değişkenlerimizi dışarıdan değiştiremiyoruz.

Backing Properties

Dışarıdan istenmeyen özellik değişikliklerini zarif bir şekilde önledik. Farklı bir teknik kullanarak , () erişirken adından sonra parantez yazmadan bir yöntem gibi davranacak bir özelliğe de ulaşabiliriz. Diğer özelliklere bağlı olarak bir değer döndüren özelliklere sahipken bunu kullanabiliriz. Kotlin'de bu tür özelliklere Backing özellikleri denir. Sınıfımızda, if (yas<18) ile kontrol ettiğimiz kısım yerine doğrudan şu şekilde `return yas >= 18` kontrol işlemi yapabiliriz. Bu şekilde değerimiz hep güncel olacaktır. Simdi resit özelliğini şu şekilde yazalım :

```
var resit: Boolean = false
    private set
    get() {
        return yas >= 18
    }
```

Not : resit özelliğimizin hala bir değişken olduğunu unutmayalım, bu değere false veya true yazabiliriz. Burada hangi değeri yazdığımızın bir önemi yoktur. Yas 18' e eşit veya büyük olduğunda resit değerimiz true olmaktadır.

Eğer değişkenlerimizde dışarıdan erişilip değiştirildiğinde tekrar bir kontrol yapmak istersek Set kısmında bu kontrolü yapabiliriz. Örneğin bir önceki kısımda dışarıdan nesne sayesinde yaşı değiştirmiştik fakat güncelleme olmadığı için yanlış sonuçla karşılaştık. Şimdi yine dışarıdan yas değiştirildiğinde doğru sonucu almak için şu şekilde yazacağız :

```
var yas = yas
     set(value){
         resit=yas>=18
         field=value // yasi value olarak ayarlar
     }
```

Bu durumda set (value) erişime açıktır. Erişimi engellemek isteseydik başına private yazabilirdik.

Nesne kullanarak yası değiştirelim.

```
fun main(arg:Array<String>){

  val ogrenci = Ogrenci( "Mert CAN" , false, 20)
  ogrenci.yas=12
  println(ogrenci)

}
```

```
Adı : Mert CAN, erkek. Yaşı : 12 ve reşit değil

Process finished with exit code 0
```

- Bu haliyle artık çıktı doğru, programımız doğru çalışmaya başladı. Son yazdıklarımızı da eklediğimizde tüm kod aşağıdaki gibidir :

Ogrenci.kt

```
class Ogrenci( isim: String,  kadin: Boolean, yas: Int) {


    var isim= isim
        private set

    var kadin = kadin
        private set

    var yas = yas
         set(value){
             resit=yas>=18
             field=value // yasi value olarak ayarlar
         }

    var resit: Boolean=false //
        private set
        get() {
            return yas >= 18
        }


    override fun toString(): String {
        var resitMi = "reşit"
        if (!resit)
            resitMi = "reşit değil"

        var cinsiyet ="kadın"
        if (!kadin)
            cinsiyet = "erkek"

        return "Adı : $isim, $cinsiyet. Yaşı : $yas ve $resitMi"
    }
}
```

Main.kt

```
fun main(arg:Array<String>){

  val ogrenci = Ogrenci( "Mert CAN" , false, 20)
  ogrenci.yas=12
  println(ogrenci)

  println("${ogrenci.resit}")
}
```

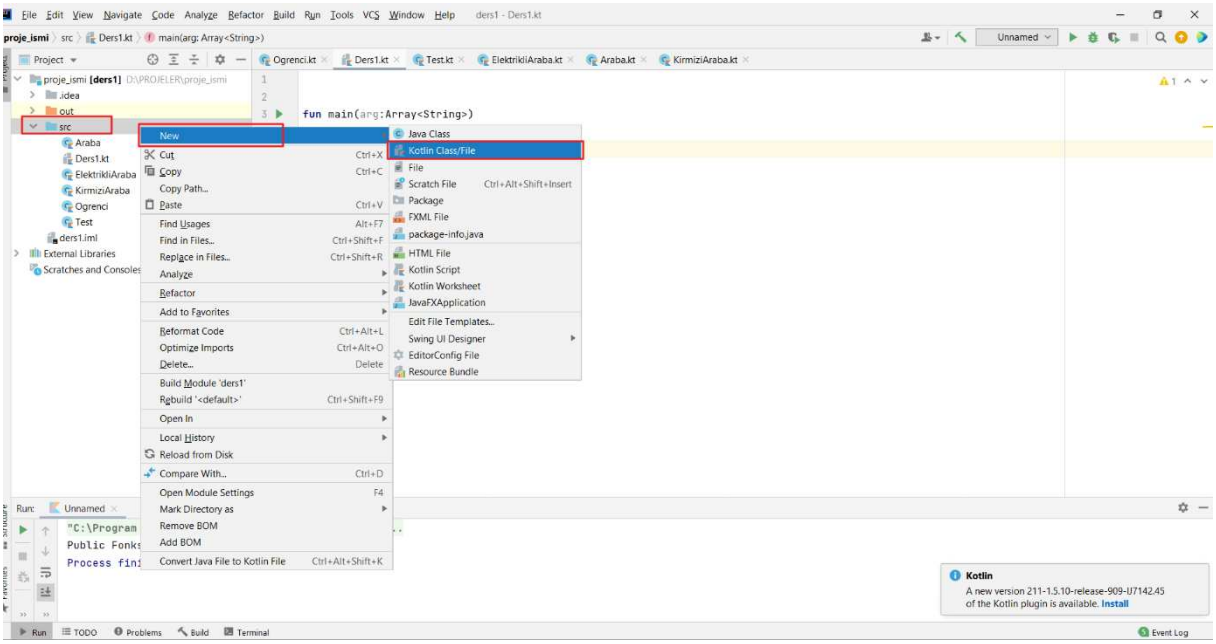
```
Adı : Mert CAN, erkek. Yaşı : 12 ve reşit değil
false

Process finished with exit code 0
```

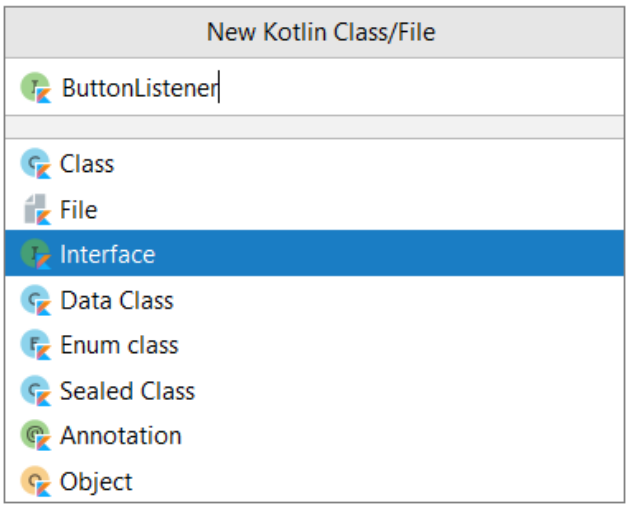
Interface (Arayüz)

Nesne tabanlı programlama içerisinde bulunan interface yani arayüzler sadece sınıfta zorunlu olarak tanımlanması gereken metotları belirtmek için kullanılır. Interface içerisinde bir imza olarak metotlarımızın adı yazılır. Yani içerisinde iş yapmayan fonksiyonlar tanımlıyoruz. Interfaceden türettiğimiz classlar bu metotları kendi içerisinde yazmak zorundadırlar. Bir bakıma planlama sürecinde kullanılıyorlar.

Interface, yazılımda sürekliliği sağlamak için çok kullanılan bir yöntemdir.

Interface Oluşturma : src > Sağ tık > New > Kotlin Class/File





- Açılan sayfada Interface'i seçip , Interface ismini yazıyoruz.
- ButtonListener adında bir interface açtık.

Interface Özellikleri

- Interface sınıfında sadece method tanımları bulunur. İçlerine kod parçacığı yazılmaz.
- İçerisinde tanımlanan method tanımları bu interface'i implemente edecek diğer sınıflar tarafından implement edilmesi zorunludur.
- Interfaceler başka bir interfaceden kalıtım alabilirler.

Örnek:

Islemler.kt

```
interface Islemler {

    fun topla(sayi1:Int, sayi2:Int)
    fun carp(sayi1: Int,sayi2:Int)
    fun karesi(sayi1:Int)

}
```

Hesapla.kt

```
class Hesapla():Islemler{

    override fun topla(sayi1:Int,sayi2:Int){
        println(sayi1+sayi2)
    }

    override fun carp(sayi1: Int, sayi2: Int) {
       println(sayi1*sayi2)
    }

    override fun karesi(sayi1: Int) {
      println(sayi1*sayi1)
    }
}
```

Ders1.kt

```
fun main(args: Array<String>) {

  val obje=Hesapla()

  obje.topla(4,5)
  obje.carp(4,5)
  obje.karesi(5)
}
```

```
9
20
25

Process finished with exit code 0
```

Soyutlama (Abstraction)

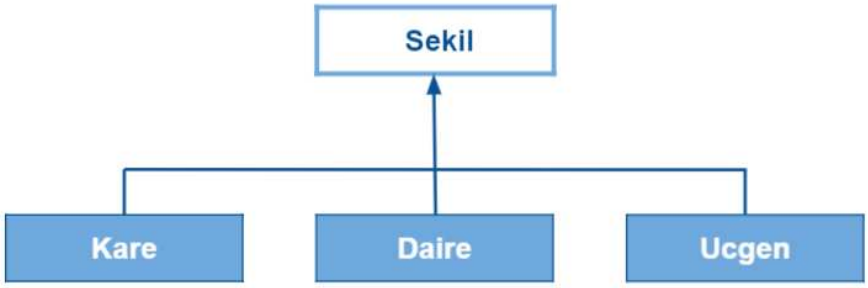
- Kalıtım yoluyla alınan özelliklerin türetilen sınıf içinde override edilerek farklı özellikler kazandırılmasıdır.
- Kotlin'de soyutlama iki farklı şekilde yapılmaktadır :
- Abstract sınıflar ve Interface kullanılarak soyutlama işlemi yapılabilir.

Soyut Sınıf (Abstract Class)

Soyut sınıflar: Diğer sınıflardan farklı olarak, soyut bir sınıf örneklenemeyen bir sınıftır. Soyut bir sınıfın nesnelerini oluşturamayız. Genellikle alt sınıflar için ortak protokolü tanımlayan bir üst sınıf olarak kullanılır. Soyut bir sınıf her zaman açıktır. Bu nedenle open anahtar kelimesini kullanmamıza gerek yoktur. Soyut sınıflar **abstract** anahtar kelimesiyle tanımlanır.

Soyut metotlar: Soyut metotlar, gövdesi olmayan metotlardır. Bu metotlar yalnızca soyut bir sınıf içinde tanımlanabilir, yani metot soyutsa bulunduğu sınıf soyut olmak zorundadır.

Daha iyi anlamak için örnekle açıklayalım:



- Diyelim ki bu sınıf hiyerarşisine sahibiz ve bu şekilleri göstermemiz gerekiyor. Ayrıca şekil nesnelerinin alanını hesaplamak için bir fonksiyonumuz olsun.
- Kare, daire veya üçgenin nasıl görüldüğünü biliyoruz. Peki ya şekil? Bir Sekil sınıfını normal bir sınıf olarak tanımlarsak, bu Sekil Classının bir nesnesi de oluşturulabilir. Ama burada bir sorunumuz var!

- Şekil genel bir isimdir şeklin nasıl göründüğünü bilmiyoruz ve dolayısıyla sekil adlı bir şeyin alanını da hesaplayamayız. Başka bir örnek vermek gerekirse mesela Canlılar sınıfı, bu da soyut bir kavramdır. Canlı ama hangi canlı? Şekil kavramı da aynen bu mantıkla düşünülebilir. Peki, kullanıcıların Sekil sınıfının nesnesini oluşturulmasına izin vermek mantıklı mıdır? Hayır.

- Bu yüzden böyle sınıfların Soyut sınıf olduklarını belirtmemiz gerekiyor. Sekil sınıfını soyut bir sınıf yapmalıyız ki, başları Sekil sınıfının nesnesini oluşturmasın. Benzer şekilde görüntüleme metodu ve alan hesaplama metodunu da soyut olduğunu belirtmeliyiz çünkü bu yöntemlerin gerçek uygulamasını bilmiyoruz.

- Sekil sınıfı, bir üst sınıf gibi davranır, alt sınıflar için bir protokol tanımlar.
- Soyut bir sınıftan miras alan sınıflar aynı protokolü takip eder, yani her alt sınıfın bir görüntüleme ve alanı hesaplamak için bir metodu vardır. Bu protokol, soyut sınıf kullanılarak uygulanır.

- Abstract(soyut) classlarda bir üyeyi (function, property) **abstract** anahtar kelimesi ile tanımlarsak bu üyeyi yalnızca alt sınıflarda override ederek kurabilir, kullanabiliriz. Eğer abstract ile işaretlemezsek bunları override etmemiz gerekmez.

Soyut Sınıf Özellikleri

- Abstract Classlar, tekrara düşen bir işi, görevi kendi bünyesinde tamamlayıp başka sınıflara devretmek için kullanılır.
- Kalıtım alınan sınıflardan en büyük farkı, Abstract classlardan instance alınamaz. Yani nesnesi üretilemez.
- Bir base yani temel sınıf oluşturup onun üzerinden işlemleri gerçekleştirmek için kullanılır.
- Abstract classlarda bir üyeyi (function, property) **abstract** anahtar kelimesi ile tanımlarsak bu üyeyi yalnızca alt sınıflarda override ederek kurabilir, kullanabiliriz. Eğer abstract ile işaretlemezsek bunları override etmemiz gerekmez.

Örnek:

```
 abstract class Sekil{

    abstract fun alan() :Double
    abstract fun goruntule()
}

 class Daire(val yariCap:Double) : Sekil(){

    override fun alan(): Double =   Math.PI * yariCap * yariCap
    override fun goruntule() {

    println("Daire görüntüleniyor")

    }
}
```

```
fun main() {
    val daire = Daire(4.0)
    println(daire.alan())
    daire.goruntule()
}
```

```
50.26548245743669
Daire görüntüleniyor

Process finished with exit code 0
```

Not : Genellikle kavramlarla ilgili sınıflar Soyut (Abstract) olarak tanımlanır. Örneğin Şekil kavramı soyuttur, Arac Sınıfı soyuttur, Canlı sınıfı soyut vb.

Örnek:

Ogrenci.kt

```
abstract class Ogrenci(var adi:String, var soyadi:String)
//Soyut sınıf oluşturma.
 {

   init{

       println("Adi: "+adi)
       println("Soyadi: "+soyadi)
   }

     //soyut olmayan fonksiyon
     fun yaz(){

         println("yaz() fonksiyonu, soyut sınıfın soyut olmayan
                  fonksiyonudur.")

     }
     //soyut fonksiyon
     abstract fun fonk(mesaj:String)
}
```

Okul.kt

```
class Okul (adi:String, soyadi:String):Ogrenci(adi,soyadi) {

//soyut sınıftan, bir sınıf türettik

    override fun fonk(mesaj: String)

    //soyut sınıfta tanımladığımız soyut fonksiyonu override ederek
burada kullanmak zorundayız.
    {
        println(mesaj)
    }
}
```

Ders1.kt

```
fun main(args: Array<String>) {

  val obje=Okul("Zeynep","Yıldız")

  obje.fonk("Ben bir öğrenciyim.")

  obje.yaz()

}
```

```
Adi: Zeynep
Soyadi: Yıldız
Ben bir öğrenciyim.
yaz() fonksiyonu, soyut sınıfın soyut olmayan fonksiyonudur.

Process finished with exit code 0
```

Çokbiçimlilik (Polymorphism)

- Polimorfizm yani çok biçimlilik, Nesne yönelimli programlamanın en önemli konularından biridir. Miras ile bağlantılıdır.
- Poly, çok anlamına gelir ve morph, formlar anlamına gelir. Yani aynı yöntem, nesneye bağlı olarak farklı davranır. Burada, nesneye bağlı olarak farklı davranışlar sergileyen yöntemin farklı biçimlerine sahip oluruz.
- Polimorfizmi basit bir şekilde şöyle düşünebiliriz: Bir ebeveyn çocuğuna bir referans tutabilir ve ebeveyn sınıfı tarafından sağlanan yöntemleri çağırabilir.

Örnek:

```
open class Sekil {
    open fun alan():Double{
        return 0.0
    }
}

class Daire(val yarıCap:Double):Sekil(){
    override fun alan():Double{
        return Math.PI*yarıCap*yarıCap
    }
}

class Kare(val kenar:Double):Sekil(){
    override fun alan():Double{
        return kenar*kenar
    }
}

fun main(arg:Array<String>){


    val daire : Sekil = Daire(4.0)
    val kare : Sekil= Kare(4.0)
}
```

- Burada temel sınıf olan ve kalıtım için açık **Sekil Sınıfı** tanımladık. Bu şekil sınıfını temel sınıf olarak kullanarak, Daire ve Kare olmak üzere iki sınıf daha tanımladık.
- Hem Daire hem de Kare, bu şekillerin alanını hesaplamanın bir yolunu tanımlayan geçersiz kılınan alan fonksiyonuna sahiptir.
- Ana fonksiyonda, bu sınıfların iki tane nesnesini tanımladık. (Daire ve Kare) Ancak Sekil sınıfının bir referansını oluşturduk. Yani Sekil sınıfı, alt sınıflarının bir referansını tutabilen bir üst sınıftır.
- Bu, alt sınıfların nesnelerini ebeveynin sınıf referansında saklayabileceğimiz polimorfizm teriminin bir parçasıdır. Basitçe, alt nesneleri ebeveynin sınıf referansında sakladık.

Şimdi bir sonraki adıma geçelim.

```
fun main(arg:Array<String>){


    val daire : Sekil = Daire(4.0)
    val kare : Sekil= Kare(4.0)

    val sekiller = arrayOf(daire, kare)
    alanHesapla(sekiller)
}

fun alanHesapla(sekiller:Array<Sekil>){

    for(sekil in sekiller)
    {
        println(sekil.alan())
    }
}
```

```
50.26548245743669
16.0


Process finished with exit code 0
```

- Yukarıda yazdığımız kodda, alanları hesaplamak için bir fonksiyon tanımladık. Sekil sınıfının nesnelerini iletebilmek için dizi tanımladık ve nesnelerin her birinde alanHesapla metodu çağrılır.

- **sekil.alan() işlevi**, şeklin ne olduğunu bilmediği için polimorfik davranış sergiliyor. Sadece alan yöntemini çağırıyor.

- Herhangi bir şekil nesnesini veya gelecekte ortaya çıkabilecek herhangi bir şekli iletebiliriz. Gelecekte uygulamamızda bu kod parçasını değiştirmemize gerek kalmaz.

- Bu yaptığımız işlem Polimorfizmdir, alan metodu nesneye bağlı olarak birçok form alıyor. Çalışma zamanında hangi alan metodunu çağırması gerektiğine karar veriyor.

- Diyelim ki gelecekte Ucgen sınıfımız da var, bu aynı kod parçasını kullanabiliriz ve herhangi bir değişikliğe ihtiyaç duymaz tamamen aynı şekilde çalışır.