14-21.10.2025 Ders Notları

Notların Kapsadığı Konular: Generic'ler (Generic Classes & Functions), Function Types, Lambda Expressions, Higher-Order Functions, Collection Processing (Filter, Fold)

Tarih: 14-21 Ekim 2023

Program: Meta Android Developer Professional Certificate

Generic'ler (Generic Classes & Functions)

- **Generic'lerin Mantığı:** Aynı kodu farklı tiplerle (String, Int, vs.) yeniden kullanabilmek için var. **Type safety** sağlarlar, yani derleme zamanında hataları yakalarlar.
- **Generic Class** yazmak normal class yazmaya benziyor, sadece `<T>` gibi bir **type parameter** ekliyorsun.

  ```
  class Item<T>(val value: T)
  ```

  Kullanırken:

  ```
  val item1 = Item(10) // Type inferred, T = Int
  val item2 = Item("Merhaba") // T = String
  ```

- **Generic Constraints:** Her tipe izin vermeyip, belirli bir üst sınıftan (parent class) türemiş tiplerle sınırlayabiliyorsun. `where T : Coffee` gibi. Bu sayede sadece `Coffee` altındaki `Latte`, `Cappuccino` gibi tipler kullanılabiliyor. Harika bir özellik.
- **Avantajları:**
  1. **Type-safety:** Sadece belirlediğin tipte nesne koyabilirsin.
  2. **Compile time checking:** Runtime'da değil, derleme anında hata verir.
  3. **No type-casting:** `(String) obj` gibi çevirimlere gerek kalmaz.
- **Generic Functions:** Fonksiyonlar da generic olabilir. Type parametresi fonksiyon adından önce yazılır.

  ```
  fun <T> addFoodItemToCart(item: T) { ... }
  ```

- **Generic'lerin Limitatif Yanları (Sınırlılıkları):**
  - **Type Erasure:** Runtime'da generic type bilgisi silinir. Yani çalışma zamanında `List<Salad>` mı yoksa `List<Beverage>` mi olduğunu bilemezsin. Hepsi `List<*>` (star projection) olarak görünür. Bu yüzden `item is List<String>` gibi bir kontrol yapamazsın.
  - **Unchecked Cast:** Type erasure'dan dolayı, yaptığın type cast'lar derleyici tarafından tam olarak kontrol edilemez ve "unchecked cast" uyarısı alırsın. Bu konuda dikkatli olmak lazım, runtime hatası yeme ihtimali var.

Function Types (Fonksiyon Tipleri)

- Fonksiyonları bir değişkene atayabildiğimiz, başka fonksiyona parametre olarak verebildiğimiz tipler.
- **Tanımlama Şekli:** `(Parametre Tipleri) -> Dönüş Tipi`
  - `(Int, String) -> String` // İki parametre alır, String döndürür.
  - `() -> String` // Parametre almaz, String döndürür.
  - `(Int) -> Unit` // Int alır, bir şey döndürmez (void gibi).
- **Instantiate Etme (Örnek Oluşturma):**
  1. **Lambda Expression:** `{ a: Int, b: Int -> a + b }`
  2. **Anonymous Function:** `fun(s: String): Int { return s.toIntOrNull() ?: 0 }`
  3. **Callable Reference (::):** Varolan bir fonksiyonu referans gösterir. `::multipliedBy2`
  4. **Bir class'a interface olarak implemente ettirme.**
- **Çağırma:** `fonksiyonDegiskeni(5)` veya `fonksiyonDegiskeni.invoke(5)` şeklinde.

Lambda Expressions

- Fonksiyonları kısa ve öz bir şekilde ifade etmenin yoludur. Süslü parantez `{}` içine yazılır.
- **Temel Kullanım:**

```
val taxCalculator = { price: Double, taxRatio: Double -> price * taxRatio }
val tax = taxCalculator(200.0, 0.065)
```

- **Extension Function Olarak Lambda:** Bir sınıfa yeni bir fonksiyon eklemek gibi düşünülebilir. Çok kullanışlı görünüyor.

```
val taxCalculator: Double.(Double) -> Double = { ratio -> this * ratio }
val tax = 200.0.taxCalculator(0.065)
```

  Buradaki `this`, `200.0` değerine (yani Double nesnesine) karşılık geliyor. Bunu daha iyi anlamak için pratik yapmak şart.

Higher-Order Functions

- Parametre olarak fonksiyon alan veya sonuç olarak fonksiyon döndüren fonksiyonlardır.
- **Kullanım Amacı:** Kod tekrarını azaltır, daha modüler ve okunabilir kod yazmayı sağlar. **Tight coupling**'i (sıkı bağlılığı) azaltır. Mesela bir class'ın tümünü değil, sadece ihtiyacın olan bir fonksiyonunu parametre olarak geçebilirsin.
- **Örnek - UI ve Click Listener:**

```
@Composable
fun SimpleUI(onClick: (Int) -> Unit) {
    var count by remember { mutableStateOf(0) }
    Button(onClick = { onClick(++count) }) {
```

```
        Text("Tıkla")
    }
}

// Kullanırken
SimpleUI { clickCount ->
    Toast.makeText(this, "Buton $clickCount kez tıklandı",
Toast.LENGTH_SHORT).show()
}
```

Bu örnekte `SimpleUI` fonksiyonu, `onClick` adında bir fonksiyon tipi bekliyor. Bu sayede butona tıklandığında ne yapılacağını dışarıdan dinamik olarak belirleyebiliyoruz. Çok temiz!

Collection Processing (Koleksiyon İşlemleri)

- Koleksiyonlar (List, Set vs.) üzerinde filtreleme, sıralama, dönüştürme gibi işlemler yapmaktır.
- **Temel Adımlar:**
  1. Koleksiyon elemanlarına eriş.
  2. Yapılacak işlemi (operation) tanımla ve uygula.
  3. Değişmiş/transform olmuş yeni koleksiyonu elde et.
- **Örnek Senaryo:** Little Lemon'ın siparişlerini işlemek. Çarşamba günkü siparişleri filtrele -> Büyükten küçüğe sırala -> Toplam sipariş tutarını hesapla.

Filter

- Bir koleksiyonu belirli bir koşula (**predicate**) göre filtreler, koşulu sağlayan elemanlardan yeni bir liste oluşturur.
- **Örnek - Yeni Çalışanları Filtrele:**

```
fun newEmployees(hiringDate: String): List<Employee> {
    return allEmployees.filter { it.hiringDate >= hiringDate }
}
```

- **Örnek - Malzemeye Göre Yemek Filtrele:**

```
fun dishesWithIngredient(ingredient: String): List<Dish> {
    return dishes.filter { ingredient in it.ingredients }
}
```

`filter` gerçekten en sık kullanacağım fonksiyonlardan biri olacak gibi duruyor.

Fold

- Bir koleksiyonu tek bir değere "indirgemek" (aggregate etmek) için kullanılır. `map` veya `filter` gibi özel amaçlı fonksiyonların yapamayacağı daha genel işlemler için birebirdir.

- **Çalışma Mantığı:** Bir `initial` başlangıç değeri ve bir `operation` lambda'sı alır. Koleksiyondaki her eleman için `operation`'ı çalıştırır ve her seferinde bir önceki adımdan kalan değeri (`accumulator`) ve o anki elemanı lambda'ya parametre olarak verir. Sonuçta `accumulator`'ın son hali döndürülür.
- **Örnek - Tüm Malzemeleri Topla (Tekilsiz - Set):**

```
fun allIngredients(): Set<String> {
    return dishes.fold(mutableSetOf()) { ingredients, dish ->
        ingredients.addAll(dish.ingredients)
        ingredients
    }
}
```

- **Örnek - Toplam Bakiye Hesaplama:**

```
fun balanceAfterOrders(orders: Collection<Double>): Double {
    return orders.fold(balance) { currentBalance, orderIncome ->
        currentBalance + orderIncome
    }
}
```

`Fold` biraz karmaşık geliyor, üzerinde biraz daha pratik yapmam lazım. `reduce` ile arasındaki farkı da araştırmalıyım.
