TÜBİTAK BİLGEM YTE | YAZILIM TEKNOLOJİLERİ ARAŞTIRMA ENSTİTÜSÜ

JAVA'DA FONKSİYONEL PROGRAMLAMA

OCAK 2023 / SAYI: 01

ARAŞTIRMA SERİSİ

Simge ve Kısaltmalar

Kısaltmalar	Açıklama
API	Application Programming Interface (Uygulama Programlama Arayüz)
SAM	Single Abstract Method (Tekil Soyut Metot)
TÜBİTAK	Türkiye Bilimsel ve Teknolojik Araştırma Kurumu
BİLGEM	Bilişim ve Bilgi Güvenliği İleri Teknolojiler Araştırma Merkezi
YTE	Yazılım Teknolojileri Araştırma Enstitüsü

Yazar: Serhat Sağlık
Uzman Araştırmacı, Yazılım Geliştirme Bölümü
TÜBİTAK BİLGEM YTE

İletişim: 0(312) 289 92 22 - yte.bilgi@tubitak.gov.tr

yte.bilgem.tubitak.gov.tr

İçindekiler

Önsöz

TÜBİTAK BİLGEM Yazılım Teknolojileri Araştırma Enstitüsü, 2012 yılından bu yana yazılım teknolojilerinde AR-GE faaliyetleri yürüten bir araştırma kuruluşudur. Araştırma faaliyetlerinde elde ettiği birikimini stratejik, hassas ve kritik projeler yürüterek kamu adına hayata geçirmekte; kurumlarımıza dijital dönüşüm, yazılım geliştirme teknolojileri ve kalite süreçleri konusunda danışmanlık vermektedir.

Araştırma Serisi ile TÜBİTAK BİLGEM YTE kurum içi çalışmaların yaygınlaştırılması ve sektörün erişimine açılması amaçlanmaktadır. Araştırma Serisi'nde yayınlanan çalışmalar TÜBİTAK BİLGEM YTE çalışanlarının projelerde elde ettiği bilgi birikimini paylaşmak adına derlenmiştir. Bu çalışmalar ile ülkemizin yazılım sektörüne katkı sağlanması hedeflenmektedir.

Giriş

Fonksiyonel programlama, fonksiyonlar kullanarak program tasarlama modelidir. Obje Tabanlı Programlama modeline göre güçlü ve zayıf olduğu yönler bulunur. Sektörde oldukça popüler ve aslında obje tabanlı olan Java programlama diline; Java 8 sürümüyle birlikte, fonksiyonel programlama konseptlerini uygulamamıza imkân tanıyan yenilikler eklenmiştir. Bu sayede hem obje tabanlı programlama hem de fonksiyonel programlamanın iyi yönleri harmanlanarak daha verimli uygulamalar tasarlanabilir hale gelmiştir.

Java'da fonksiyonel programlama araçları; pratik işlevler yazmada, okumada oldukça kolaylık sağlamaktadır. Karmaşık döngüler Stream API sayesinde sade bir hale getirilebilir; birçok yerden çağırılabilecek fonksiyonel arayüzler kullanılarak bazı kontroller ve işlevler; kolay okunabilir, debug ve test edilebilir, yönetilebilir şekilde yazılabilmektedir.

Bu faydaların her birinin çalışma prensibi ayrı ayrı açıklanmış ve anlaşılması için kod örnekleri ve çıktıları belirtilmiştir. Güncel sürümü kullanma imkânı olmayan öğrenci ve çalışanlar için yarar sağlayacağı düşünülerek bu çalışma hazırlanmıştır.

Türkçe bir kaynak yazılması ve araştırma dolayısı ile artan bilgi birikiminin, sektördeki çalışanlar arasındaki etkileşimle yayılarak bilgi hazinesinin bir parçası haline gelmesi, yeni başlayan yazılımcıların entegrasyonunu kolaylaştıracak ve eski çalışanları daha fazla öğrenmeye teşvik edecektir.

Bu çalışma Java 8'de eklenen yenilikler olan fonksiyonel arayüzleri, lamda işaretini ve Stream API'yi kapsamaktadır. Fonksiyonel programlama tanımlanacak, fonksiyonel programlama ve obje tabanlı programlamanın farklarından bahsedilecek, Java'ya yeni eklenmiş özellikler ve bu özellikleri kullanarak fonksiyonel programlama mantığına uygun işlemler yapmak için örnekler listelenecektir. Bu bağlamda; fonksiyonel programlama hakkında farkındalık yaratarak Java dilinde fonksiyonel programlama mantığına uygun kodlar yazmayı öğrenmek ve uygulamak isteyenler için kolay anlaşılabilir bir kaynak sunma hedeflemektedir.

Java'da Fonksiyonel Programlama

1. Fonksiyonel Programlama Nedir?

Fonksiyonel programlama, bir uygulamanın verilerini ve durumunu değiştirmeden tıpkı bir matematik fonksiyonu gibi bir girdi alıp çıktı üreten fonksiyonlar kullanılarak yapılan programlamadır. Prensip olarak kökenini 1930'lu yıllarda Alonzo Church tarafından tanımlanmış lamda kalkülüs modelinden alır. Bu modele göre örneğin hipotenüs(x,y) = $\sqrt{x^2 + y^2}$ fonksiyonu isimsiz bir şekilde (x,y) => $\sqrt{x^2 + y^2}$ olarak yazılır.

2. Fonksiyonel Programlama – Obje Tabanlı Programlama Farkları

Fonksiyonel Programlama	Obje Tabanlı Programlama
• Değişmez (immutable) veri kullanır.	• Bir kısıtlaması bulunmaz.
• "Ne yapar" odaklı bildirimsel (declarative) programlama modeli kullanır.	• "Nasıl yapar" odaklı zorunlu (imperative) programlama modeli kullanır.
• Yinelemeli (iterative) veriler için özyineleme (recursive) kullanır.	• Yinelemeli veriler için döngü (loop) kullanır.
• Değişkenler ve fonksiyonlar temel öğeleridir.	• Nesneler ve modeller temel öğeleridir.
• Çok farklı işlemler yapılan az iş için kullanılır.	• Az işlem yapılan çok farklı işler için kullanılır.
• Paralel programlamayı destekler.	• Paralel programlamayı doğrudan desteklemez.
• Veri gizlemeyi desteklemez.	• Veri gizlemeyi destekler.
• İfadeler (statement) herhangi bir sırada çalışabilir.	• İfadeler belli bir sırada çalışır.

Tablodan görülebileceği gibi iki programlama tipi de farklı bir yaklaşımı benimsemektedir. Obje tabanlı programlamada (OOP) var olan objeleri/metotları kullanarak yeni objeler ve metotlar yaratmak kolaydır. Var olan işlemleri kullanarak yeni veri tiplerine kolay adapte olur ve kalıtım (inheritance) burada büyük rol oynar. Bu sebeplerle, uzun süreli devam eden ve gittikçe karmaşıklaşan projeler için daha uygundur. Fonksiyonel programlama (FP) sabit bir veri üzerine yeni işlemler eklemek için iyidir ve eski işlemler aynı şekilde kullanılmaya devam eder. Ancak, veri tipi değişikliklerine daha zor adapte olur.

Obje tabanlı bir dil olan Java, fonksiyonel programlamaya uygun olmasa da Java 8 sürümüyle birlikte lamda fonksiyonları ve fonksiyonel arayüzlerin (interface) eklenmesiyle fonksiyonel programlamanın faydalarından yararlanılmasına imkan vermektedir.

3. Fonksiyonel Arayüzler

Fonksiyonel arayüz (interface), yalnızca bir soyut (abstract) metoda sahip arayüzlere denir. Tekil Soyut Metot Arayüz (Single Abstract Method Interface - SAM) ismiyle de bilinirler. Dolayısıyla tek bir iş yaparlar. Java 8 ile birlikte eklenmiş olan lamda fonksiyonları, SAM arayüzleri için kullanılabilir.

```
public class Main {
    public static void main(String args[]) {
        eskiYontem();
        yeniYontem();
    }

    public static void eskiYontem() {
        new Thread(new Runnable() {
            @Override public void run() {
                System.out.println("Klasik yontemle thread yaratma" ) ;
            }
        }).start();
    }

    public static void yeniYontem() {
        new Thread(() → {
            System.out.println("Yeni yontemle thread yaratma");
        }).start();
    }
}
```

Örnekteki kod Runnable interface'ini iki yöntemle de kullanarak iki tane thread yaratacak ve çıktı olarak şunu üretecektir:

Klasik yontemle thread yaratma

Yeni yontemle thread yaratma

Bir arayüzün fonksiyonel arayüz olduğunu belirtmek için @FunctionalInterface etiketi (annotation) kullanılır. Metot tanımında **abstract** terimi (keyword) kullanılmak zorunda değildir, çünkü zaten ön tanımlı olarak abstract olmalıdır. Örnek olarak **TestFonksiyonelArayüz** arayüzü verilebilir.

```
@FunctionalInterface
public interface TestFonksiyonelArayüz {
    int birSeyYap();
}
```

@FunctionalInterface'den dolayı eğer ikinci bir metot eklenseydi, program çalıştırıldığında derleyici (compiler) **TestHataliFonksiyonelArayüz** arayüzünün fonksiyonel arayüz tanımına uymadığını belirten aşağıdaki hatayı atacaktı ve program çalışmayacaktı.

```
@FunctionalInterface
public interface TestHataliFonksiyonelArayüz {
    int birSeyYap();
    int baskaBirSeyYap();
}
```

```
java: Unexpected @FunctionalInterface annotation
  TestHataliFonksiyonelArayüz is not a functional interface
    multiple non-overriding abstract methods found in interface
TestHataliFonksiyonelArayüz
```

Dört farklı fonksiyonel arayüz tipinden söz edilebilir:

1-) Consumer

2-) Supplier

3-) Predicate

4-) Function

3.1. Consumer

Consumer tüketici anlamına gelir ve bu arayüzler tek bir argüman (argument) alıp hiçbir değer dönmezler. Örnek kullanım:

```
public class Main {
    static Consumer<Character> charConsumer = ch → System.out.println(ch);

    public static void main(String args[]) {
        Character[] charArray = {'t', 'e', 's', 't'};
        Arrays.stream(charArray).forEach(charConsumer);
    }
}
```

Bu program bize aşağıdaki çıktıyı verecektir:

```
t
e
s
t
```

Consumer arayüzüne benzer olarak bir yerine iki argüman alan Bi-Consumer arayüzü de bulunmaktadır.

```
public static void main(String[] args) {
    Map<String, Integer> harcliklar = Map.of(“mehmet”,100, “ahmet”, 200);
    BiConsumer<String, Integer> biConsumer = (kisi, harclik) → System.out.
println(kisi + “’in harçlığı: “ + harclik);
    harcliklar.forEach(biConsumer);
}
```

Örneğin, bu consumer map içindeki kişi ve sayı değerlerini alarak aşağıdaki çıktıyı verir:

```
ahmet’in harçlığı: 200
mehmet’in harçlığı: 100
```

Ön tanımlı olarak ikiden fazla argüman alan tipi bulunmaz. Fakat argümanlardan birisi BiConsumer olan farklı bir BiConsumer kullanılarak, üç argümanlı veya benzer şekilde daha fazla argüman alan çeşitleri tanımlanabilir.

Consumerların çalıştıktan sonra verileri tükettiklerinden ve hiçbir değer döndürmediklerinden bahsettik fakat birden fazla işlem yapmaya ihtiyaç varsa consumerları ardı ardına çalışacak şekilde bağlamamıza imkân tanıyan andThen kullanılabilir.

Örneğin aşağıdaki kodda her bir liste elemanı için ilk önce elemanın kendisini çıktı veren consumer, ardından da elemanın bir fazlasının çıktısını veren consumer çalışacak ve sırasıyla 1,2,3,4,5,6 çıktıları alınmış olacaktır.

```
Consumer<Integer> consumer1 = i → System.out.println(i);
Consumer<Integer> consumer2 = i → System.out.println(i+1);
List<Integer> sayiList = Arrays.asList(1, 3, 5);

sayiList.stream().forEach(i → consumer1.andThen(consumer2).accept(i));
sayiList.stream().forEach(consumer1.andThen(consumer2));
```

İki farklı kullanımı gösterilmiş andThen örnekleri aynı şekilde çalışmaktadır ve ikisi de ayrı olarak aşağıdaki çıktıyı verir:

```
1
2
3
4
5
6
```

3.2. Supplier

Supplier tedarikçi anlamına gelir ve bu arayüzler mantık olarak consumer'ın tersidir denebilir. Argüman almazlar fakat çıktı üretirler. Bu arayüz içinde tanımlı olarak sadece **get** metodu bulunur.

```
public class Main {
    static Supplier<String> messageSupplier = () → "Test Mesaji";

    public static void main(String args[]) {
        System.out.println(messageSupplier.get());
    }
}
```

Örneğin, bu örnekte çıktımız **messageSupplier** tarafından dönülen string'in kendisi olacaktır:

```
Test Mesaji
```

Supplier argüman almadığı için BiSupplier gibi versiyonları vardır, fakat belli tipleri dönen versiyonları bulunmaktadır. Örneğin: IntSupplier, LongSupplier, BooleanSupplier, DoubleSupplier.

IntSupplier için örnek kullanım şu şekilde gösterilebilir:

```
public class Main {
    static IntSupplier birdenOnaSayiSupplier =
            () → (int) (Math.random() * 10 + 1);

    public static void main(String args[]) {
        System.out.println(birdenOnaSayiSupplier.getAsInt());
    }
}
```

Bu supplier bize 1-10 arasında rastgele bir sayı değeri dönecektir. Örnekten görülebileceği gibi bu arayüzler **get** metodu yerine **getAsX** şeklinde metotlar kullanır.

3.3. Predicate

Predicate dayanak gibi bir anlama gelir ve bu arayüzler bir argüman alıp üzerinde kontrol mantığı çalıştırıp boolean değer dönerler. Bu arayüz içinde test metodu bulunur ve argüman üzerinde kontrol mantığını test eder.

```
public class Main {
    static Predicate<Integer> isCiftSayi = (i) → i % 2 == 0;

    public static void main(String args[]) {
        System.out.println(isCiftSayi.test(10));
        System.out.println(isCiftSayi.test(9));
    }
}
```

Yukarıdaki örnekte predicate sayıların çift mi tek mi olduğunu kontrol eder ve istenilen int değeri üzerinde bunu test eder. Örnekteki çıktı aşağıdaki şekilde olacaktır:

```
true
false
```

Consumer'da olduğu gibi predicate için de iki argüman alan BiPredicate versiyonu bulunur.

```
public class Main {
    static BiPredicate<Integer, Integer> esitMi = (x, y) → x == y;

    public static void main(String args[]) {
        System.out.println(esitMi.test(10, 10));
        System.out.println(esitMi.test(11, 9));
    }
}
```

Bu predicate sayı değerlerini karşılaştırarak eşitlik durumunu döner. Çıktımız şu şekilde olacaktır:

```
true
false
```

Bu arayüzün de IntPredicate, LongPredicate gibi tipe özel çeşitleri bulunmaktadır. İlk örnek olan isCiftSayi predicate arayüzü doğrudan IntPredicate olarak aşağıdaki şekilde yazılabilir:

```
public class Main {
    static IntPredicate isCiftSayi = (i) → i % 2 == 0;

    public static void main(String args[]) {
        System.out.println(isCiftSayi.test(10));
        System.out.println(isCiftSayi.test(9));
    }
}
```

Bu aynı çıktıyı verecektir:

```
true
false
```

Birden fazla predicate kullanarak yeni bir kontrol mantığı oluşturmak istenirse and, or, not, negate metotları kullanılabilir.

Örneğin bir değerin çift sayı olup, altıya eşit olmadığını test edelim. Bunun için çift sayı olup olmadığını kontrol eden bir predicate ve altıya eşit olup olmadığını kontrol eden ikinci bir predicate tanımlayıp aşağıdaki gibi ikinci kontrolün değili alınarak kullanılabilir:

```
Predicate<Integer> ciftSayi = sayi → sayi % 2 == 0;
Predicate<Integer> altiyaEsit = sayi → sayi == 6;
Predicate<Integer> ciftSayiVeAltiyaEsitDegil = ciftSayi.and(altiyaEsit.
negate());

System.out.println(ciftSayiVeAltiyaEsitDegil.test(3));
System.out.println(ciftSayiVeAltiyaEsitDegil.test(6));
System.out.println(ciftSayiVeAltiyaEsitDegil.test(4));
```

Bu kontrolün çıktısı da aşağıdaki gibi olacaktır:

```
false
false
true
```

3.4. Function

Function fonksiyon demektir ve bu fonksiyonel arayüz tipi matematikteki fonksiyon gibi düşünülebilir. Argüman üzerinde gerekli işlemleri yapıp herhangi bir tipte veri döner. Arayüzde tanımlı **apply** metodu ile argüman fonksiyona sokulabilir. Kendi dışındaki hiçbir değişkenin durumunu etkilemeyen ve kendi dışındaki şeylerden etkilenmeyen fonksiyonlara saf (pure) fonksiyon denir. Character tipinde veri alıp işlem sonrası Integer veri dönen örnek bir fonksiyon şu şekilde tanımlanabilir:

```
public class Main {
    static Function<Character, Integer> karakterinIntDegeriniGetir =
            (ch) → Character.getNumericValue(ch);

    public static void main(String args[]) {
        System.out.println(karakterinIntDegeriniGetir.apply('a'));
        System.out.println(karakterinIntDegeriniGetir.apply('b'));
    }
}
```

Bu çıktı olarak karakterlerin Integer değerleri olan aşağıdaki çıktıyı verecektir:

```
10
11
```

Bu arayüzün de iki argümanlı hali olan BiFunction öntanımlı olarak bulunmaktadır. BiConsumer için verilen örnekte konsola yazdırılan String değerini, doğrudan dönen bir BiFunction şu şekilde tanımlanabilir ve kullanılabilir:

```
public static void main(String args[]) {
    BiFunction<String, Integer, String> charConsumerFunction = (str, i) → str +
"'in harçlığı: " + i;

    Map<String, Integer> harcliklar = Map.of("mehmet",100, "ahmet", 200);
    harcliklar.entrySet().forEach(
            mapEntry → System.out.println(
                    charConsumerFunction.apply(mapEntry.getKey(), mapEntry.
getValue())));
}
```

Bu durumda da kod aynı çıktıyı verecektir:

```
ahmet'in harçlığı: 200
mehmet'in harçlığı: 100
```

Bu arayüzün de IntFunction, LongFunction gibi tipe özel çeşitleri bulunmaktadır. Predicate için verilen isCiftSayi örneği argüman olarak Integer kullanılacağından, IntFunction olarak ve boolean değer dönecek şekilde aşağıdaki gibi tanımlanabilir:

```
public class Main {
    static IntFunction<Boolean> isCiftSayi = i → i % 2 == 0;

    public static void main(String args[]) {
        System.out.println(isCiftSayi.apply(10));
        System.out.println(isCiftSayi.apply(9));
    }
}
```

Bu durumda da yine çıktıyı verecektir:

```
true
false
```

Fonksiyonlar için ek olarak UnaryOperator ve BinaryOperator adlı iki arayüz bulunmaktadır. Bunlardan UnaryOperator, Function arayüzünden; BinaryOperator ise BiFunction arayüzünden extend eder. Farkları girdi ve çıktı tiplerinin aynı olmasıdır. Örneğin, iki sayının birbirine bölümünden kalan sayıyı bulmak için bir BinaryOperator şu şekilde tanımlanabilir:

```
public class Main {
    static BinaryOperator<Integer> bolumdenKalan = (x, y) → x % y;

    public static void main(String args[]) {
        System.out.println(bolumdenKalan.apply(10, 3));
    }
}
```

Çıktı da aşağıdaki gibi olacaktır:

```
1
```

Ayrıca, üç argüman alan TriFunction gibi tipleri tanımlamak da mümkündür:

```
@FunctionalInterface
public interface TriFunction {
    int apply(int x, int y, int z);
}

public class Main {
    public static void main(String args[]) {
        TriFunction ucunuCarp = (x,y,z) → x * y * z;
        System.out.println(ucunuCarp.apply(10, 3, 5));
    }
}
```

Bu şekilde üç adet int değeri alıp int değeri dönen bir TriFunction tanımlanarak, yapılacak işlem lamda fonksiyonu olarak kullanılan yerde tanımlanabilir ya da implementasyonu da yapıp aşağıdaki gibi çağırılabilir:

```
public class UcSayiyiCarp implements TriFunction {
    @Override
    public int apply(int x, int y, int z) {
        return x * y * z;
    }
}

public class Main {
    public static void main(String args[]) {
        UcSayiyiCarp ucSayiyiCarp = new UcSayiyiCarp();
        System.out.println(ucSayiyiCarp.apply(10, 3, 5));
    }
}
```

Alternatif olarak; fonksiyon içinde fonksiyon kullanmak istenirse, şu şekilde bir kullanım da olabilir:

```
@FunctionalInterface
public interface NestedFunction {
    Integer apply(BiFunction<Integer, Integer, Integer> f,
                  Integer x, Integer y, Integer z);
}

public class NestedFunctionImpl implements NestedFunction {
    @Override
    public Integer apply(BiFunction<Integer, Integer, Integer> f,
                         Integer x, Integer y, Integer z) {
        return f.apply(x,y) * z;
    }
}

public class Main {
    public static void main(String args[]) {
        BiFunction<Integer, Integer, Integer> function =
                (x, y) → x * y;

        NestedFunction nestedFunction = new NestedFunctionImpl();
        System.out.println(nestedFunction.apply(function, 10, 3, 5));
    }
}
```

Her değerin ayrı ayrı apply edildiği, bir üçlü çarpım metodu şu şekilde de tanımlanabilir:

```
public class Main {
    public static void main(String args[]) {
        Function<Integer,
                Function<Integer,
                        Function<Integer, Integer>>> ucluCarpim
                = x → y → z → x * y * z;
        System.out.println(ucluCarpim.apply(10).apply(3).apply(5));
    }
}
```

Bu yönteme Currying denmekle birlikte, dört yöntemin de çıktısı şu şekilde olacaktır:

```
150
```

Eğer functionların basit bir şekilde birbiri ardına çalışmalarını sağlamak isteniyorsa andThen burada da kullanılabilir.

Örneğin aşağıdaki kodda yedi çıkarma ve ikiyle çarpma için iki farklı function tanımlanmış. Bunları art arda çalıştıracak yeni bir fonksiyonu andThen kullanarak tanımlanabilir. Burada ilk önce andThen'in çağrıldığı function çalışır ardından andThen'e argüman olarak geçilen function çalışır.

```
Function<Integer, Integer> yediCikar = sayi → sayi - 7;
Function<Integer, Integer> ikiyleCarp = sayi → sayi * 2;
Function<Integer, Integer> yediCikarVeIkiyleCarp = yediCikar.
andThen(ikiyleCarp);

System.out.println(yediCikarVeIkiyleCarp.apply(10));
```

Çıktı aşağıdaki gibi olacaktır:

```
6
```

Function için ek olarak compose metodu da bulunmaktadır. Ve andThen'in tersi sırasıyla çalışır. Yani ilk önce compose'a argüman olarak geçilen function, ardından ise compose'un çağrıldığı function çalışır. Compose ile aynı çıktıyı almak için aşağıdaki gibi tanımlanması gerekirdi.

```
Function<Integer, Integer> yediCikar = sayi → sayi - 7;
Function<Integer, Integer> ikiyleCarp = sayi → sayi * 2;
Function<Integer, Integer> yediCikarVeIkiyleCarp = ikiyleCarp.
compose(yediCikar);

System.out.println(yediCikarVeIkiyleCarp.apply(10));
```

4. Stream API

Stream API, Java 8 ile eklenmiş bir yeniliktir. Stream akıntı anlamına gelir ve çalışma mantığı olarak da bir akıntı üzerine gelen verileri sırayla ya da paralel işleyen metotlar olarak düşünülebilir. Stream sayesinde loop kullanan imperative metotlar, fonksiyonel programlamaya uygun şekilde değiştirilebilmektedir.

```
public class Main {
    public static void main(String args[]) {
        List<Integer> integerList = Arrays.asList(1,2,3,4,5);
        int result = 0;
        for (int i = 0; i < integerList.size(); i++) {
            result += integerList.get(i) * 2;
        }
        System.out.println(result);

        result = integerList.stream()
                .map(i → i * 2)
                .reduce(0, Integer::sum);
        System.out.println(result);
    }
}
```

Bu örnekte **::** sembolünün **Integer.sum()** metodu referansı olarak da kullanılması gösterilmiştir.

Loop kullanırken sürekli değeri güncellenen result kullanarak, stream örneğinde immutable mantığına da uyum sağlamıştır. Bu örnekte alınan çıktı ikisi için de aynı olacaktı fakat, toplamaya 0'dan değil de farklı bir sayıdan başlanması istenseydi result değerine başka bir sayı atamanın karşılığı stream için reduce içindeki 0'ı artırmak olacaktı. Stream'in başlıca işlevlerinden aşağıda kısaca bahsedilmiştir:

4.1. Map

Map operatörü stream içindeki her eleman üzerinde belirtilmiş fonksiyonu çağırarak yeni bir stream yaratır. Yukarıdaki örnekte 1, 2, 3, 4, 5 elemanlarını bulunduran streamden 2, 4, 6, 8, 10 elemanlarını bulunduran yeni bir stream yarattı.

4.2. Filter

Filter operatörü stream içindeki verileri belirtilmiş koşula göre test ederek, sadece uyan verilere sahip yeni bir stream yaratır. Örneğin, aşağıdaki örnekte filter sonucundaki stream sadece çift sayıları bulunduran {2,4} olacaktır:

```
List<Integer> integerList = Arrays.asList(1,2,3,4,5);

List<Integer> streamedList = integerList.stream()
        .filter(i → i % 2 == 0)
        .collect(Collectors.toList());
```

4.3. Collect

Stream verilerini farklı şekillerde gruplayıp belirtilen birimde dönen operatördür. Önceki örnekte liste olarak dönmesi söylenmiştir.

4.4. ForEach

ForEach bir consumerdır ve streamde bulunan her veri üzerinde forEach içinde belirtilmiş işlevi çağırır. Örneğin aşağıda listenin her elemanı için, elemanın iki katını yazdıran bir kullanım mevcuttur.

```
List<Integer> integerList = Arrays.asList(1,2,3);

integerList.forEach(integer → System.out.println(integer * 2));
```

Program çıktısı:

```
2
4
6
```

4.5. AllMatch

AllMatch operatörü stream içindeki verileri belirtilmiş koşula göre test eder ve eğer tüm veriler koşulu sağlıyorsa true, herhangi biri bile sağlamıyorsa false döner.

Örneğin aşağıda yer alan sayiList listesinde tek sayılar da bulunduğu için allMatch(çift sayı mı) kontrolü false dönecektir. Fakat ciftSayiList içindeki tüm sayılar çift olduğu için true dönecektir.

```
List<Integer> sayiList = Arrays.asList(1,2,3);
List<Integer> ciftSayiList = Arrays.asList(2,4,6);

System.out.println(sayiList.stream().allMatch(sayi → sayi % 2 == 0));
System.out.println(ciftSayiList.stream().allMatch(sayi → sayi % 2 == 0));
```

4.6. AnyMatch

AnyMatch operatörü stream içindeki verileri belirtilmiş koşula göre test eder ve eğer hiçbir veri koşulu sağlamıyorsa false, herhangi biri bile sağlıyorsa true döner.

4.5'teki örneğin bir benzerini burada da kullanılabilir. Fakat burada elemanlardan herhangi birinin koşulu sağlaması yeterli olduğu için sayiList ve ciftSayiList true, içinde hiç çift sayı bulundurmayan tekSayiList false dönecektir.

```
List<Integer> sayiList = Arrays.asList(1,2,3);
List<Integer> ciftSayiList = Arrays.asList(2,4,6);
List<Integer> tekSayiList = Arrays.asList(1, 3, 5);

System.out.println(sayiList.stream().anyMatch(sayi → sayi % 2 == 0));
System.out.println(ciftSayiList.stream().anyMatch(sayi → sayi % 2 == 0));
System.out.println(tekSayiList.stream().anyMatch(sayi → sayi % 2 == 0));
```

4.7. Distinct

Stream içindeki verileri birbiriyle karşılaştırarak, eşitlik koşulunu sağlayan çoklu verileri teke indirger.

Örneğin aşağıdaki sayiList içinde tekrar eden elemanlar bulunmaktadır. Eğer distinct kullanmadan forEach kullanıp printleseydi her eleman için bir tane olmak üzere yedi adet çıktı olacaktı. Fakat önce distinct elemanları alınıp kullanıldığı zaman tekrar eden elemanları çıkartıp, geriye kalan (1, 2, 3, 4) listesi üzerinde işlem yapacak ve dört adet çıktı verecektir.

```
List<Integer> sayiList = Arrays.asList(1,2,3,3,1,3,4);


sayiList.stream().distinct().forEach(sayi → System.out.println(sayi));
```

Program çıktısı:

```
1
2
3
4
```

4.8. Count

Stream içindeki veri sayısını döner.

4.7'de bulunan örnek buraya da uyarlanabilir. Program ilk önce listenin orijinal halinin eleman sayısını ardından da distinct çağrısından sonraki eleman sayısını çıktı olarak verecektir.

```
List<Integer> sayiList = Arrays.asList(1,2,3,3,1,3,4);

System.out.println(sayiList.stream().count());
System.out.println(sayiList.stream().distinct().count());
```

Program çıktısı:

```
7
4
```

4.9. Min/Max

Sırasıyla stream içindeki en küçük ve en büyük veriyi döner. Karşılaştırma metodu özel olarak tanımlanabilir.

Örneğin aşağıdaki liste sırasıyla 1 ve 5 çıktısı verecektir.

```
List<Integer> sayiList = Arrays.asList(3, 1, 4, 5, 2);

System.out.println(sayiList.stream().min((x, y) → x.compareTo(y)).get());
System.out.println(sayiList.stream().max((x, y) → x.compareTo(y)).get());
```

4.10. FlatMap

FlatMap operatörü map'e oldukça benzer olmakla birlikte farkı, kendi içinde streamleri otomatik olarak birleştirip tek bir stream haline getirmesidir. Buna düzleştirme (flatten) denir.

```
public class Main {
     public static void main(String args[]) {
           List<List<Integer>> integerList = Arrays.asList(Arrays.asList(1),
                 Arrays.asList(2), Arrays.asList(3));

           List<Integer> streamedList = integerList.stream()
                 .flatMap(list → list.stream())
                 .toList()
           System.out.println(streamedList);
     }
}
```

Örneğin, burada liste içindeki liste elemanlarının stream'i kolaylıkla tek bir listeye toplanabilmiştir.

4.11 Parallel

Normalde stream tek sıra halinde çalışır. Stream paralel yapıldığında ise mevcut stream paralel olarak aynı anda çalışacak küçük streamlere bölünür. Paralellik dolayısıyla verileri işleme sırası karışabilir. Bu sebeple paralel stream kullanılacağı zaman işlem sırasının değişkenliğinin sorun yaratmayacağından emin olunması gerekir.

Aşağıdaki örnekte aynı liste üzerinde önce stream oluşturulup sonradan paralele çevirme ve en baştan paralel stream oluşturma şeklinde iki kullanım görülebilir. Burada sadece, her bir stream için konsola 1, 2, 3, 4 ve 5'in basılacağından emin olunabilir. Hangi sırayla basılacağı değişkenlik gösterecektir. İki stream'in sırası bile birbirinden farklı olabilir. Örneğin birisi 3, 5, 4, 2, 1 sırasıyla çıktı verirken diğeri 3, 2, 4, 1, 5 şeklinde verebilir.

```
List<Integer> sayiList = Arrays.asList(1, 2, 3, 4, 5);

sayiList.parallelStream().forEach(i → System.out.println(i));
sayiList.stream().parallel().forEach(i → System.out.println(i));
```

Sonuç ve Öneriler

Bu çalışmada kullanılan yöntemler kolay anlaşılabilir ve düzenlenebilir olduğundan özellikle konuya giriş yapanlar için oldukça yararlı olacaktır. Çalışmanın Türkçe olması ise İngilizce kaynakların ağırlıkta olduğu ülkemiz yazılım sektörü açısından önem arz etmektedir. Gelecek çalışmalarda, fonksiyonel programlamanın kullanılmasının fayda sağlayacağı bir problem üzerine yoğunlaşılabilir. Java'da fonksiyonel programlama ile daha ileri seviyede ilgilenenler açık kaynak bir kütüphane olan **Vavr kütüphanesini** inceleyebilirler.

Referanslar

1. Sheehan, L. (n.d.). Learning functional programming in go. Retrieved November 14, 2022, https://www.oreilly.com/library/view/learning-functional-programming/9781787281394/6931699b-4fad-43fd-b201-85d104c25222.xhtml

2. Pedamkar, P. (2022, June 07). Functional programming vs OOP: Top 8 useful differences to know. Retrieved November 14, 2022, https://www.educba.com/functional-programming-vs-oop/

3. Meriç, T., & Bi, P. (2006). JFun: Functional Programming in Java. http://tekin.mericli.com/files/Mericli-2006jfun.pdf

4. https://www.vavr.io/

TÜBİTAK

BİLGEM

YTE | YAZILIM TEKNOLOJİLERİ ARAŞTIRMA ENSTİTÜSÜ

İşçi Blokları Mahallesi Muhsin Yazıcıoğlu Caddesi No:51/C 06530 Çankaya/ANKARA

+90 (312) 289 92 22 - yte.bilgem@tubitak.gov.tr

TÜBİTAK - BİLGEM Yazılım Teknolojileri Araştırma Enstitüsü (YTE)

yte.bilgem.tubitak.gov.tr