```
0(532) 315-23-79 numarası desene uyuyor.
0(554) 289-213-124 numarası desene UYMUYOR.
0531981-66-34 numarası desene UYMUYOR.
0(567) 144-78-63 numarası desene uyuyor.
```

LAMBDA İFADELERİ

JDK 8 ile dile eklenen lambda ifadeleri Java'yı temelden etkilemiştir. Bir lambda ifadesi, en basit tanımıyla, isimsiz (anonim) bir metottur. Fakat bu metotları tanımlayabilmek için **fonksiyonel arayüz** (*functional interface*) oluşturmak gerekir. Lambda ifadelerini daha iyi anlayabilmek için, fonksiyonel arayüzleri anlatmadan önce **anonim sınıf**lardan (*anonymous class*) bahsedelim.

Anonim sınıflar

Bir arayüz veya soyut sınıf üzerinden tanımlanan ve ismi olmayan sınıflara anonim sınıf denir. Bu sınıflar oluşturuldukları anda kullanılırlar. Anonim sınıfların bir örneğini alamazsınız; çünkü isimleri yoktur.

Şimdi anonim sınıfları anlayabilmek için bir örnek yapalım. Öncelikle bir arayüz tanımlayalım:

```
public interface Operation
{

    int operate(int x, int y);

}
```

Operation isminde bir arayüz tanımladık ve içine *operate()* adında bir metot yazdık. Bu metot int türünde iki sayıyı parametre olarak alır, bu sayılar üzerinde bir işlem yapar ve sonucu yine int türünde döndürür.

Şimdi bu arayüzü kullanan bir metot yazalım:

```
public class Math
{

    public static int operateTwoNumbers(int x, int y, Operation
        operation)
    {
        return operation.operate(x, y);
    }

}
```

Math adında bir sınıf tanımladık ve bu sınıf içinde statik *operateTwoNumbers()* metodunu yazdık. Bu metot int türünde iki sayıyı parametre alıyor ve bu sayılar üzerinde bir işlem yapıyor; fakat bu işlemin nasıl yapılacağını yine parametre olarak aldığımız *Operation* türündeki nesneden öğreniyoruz.

Şimdi, iki sayıyı toplayan bir *Operation* sınıfı yazalım:

```
public class AdditionOperation implements Operation
{

    @Override
    public int operate(int x, int y)
    {
        return x + y;
    }

}
```

Şimdi aşağıdaki işlemi gerçekleştirebiliriz:

```
int result = Math.operateTwoNumbers(5, 10, new AdditionOperation());
System.out.println(result);
```

Bu kodu çalıştırdığınızda konsola 15 yazar; çünkü *AdditionOperation* sınıfı toplama işlemi yapmaktadır. Fakat burada şunu düşünelim: diyelim ki, bu metodu sadece bir kez kullandık. Sadece bir kez kullandığımız bu metot için *AdditionOperation* adında bir sınıf yazmamıza gerek var mıdır?

Bunun yerine anonim sınıfları kullanabilirdik. Şimdi yukarıdaki kodu anonim sınıf kullanarak tekrar yazalım:

```
int result = Math.operateTwoNumbers(5, 10, new Operation()
{
    @Override
    public int operate(int x, int y)
    {
        return x + y;
    }
});

System.out.println(result);
```

Kalın olarak belirttiğimiz kod anonim sınıf kodudur. Gördüğünüz gibi, new deyimini kullanarak *Operation* türünde bir nesne oluşturduk; fakat bildiğiniz gibi, new deyimini arayüzler üzerinde kullanamayız, yalnızca sınıflar üzerinde kullanabiliriz. Burada da new deyimini kullanarak bir sınıf oluşturduk; fakat bu sınıfın bir ismi yoktur. Bu sınıfla ilgili bildiğimiz tek şey, *Operation* arayüzünü uyguladığıdır, dolayısıyla *operate()* metodunu yazmak zorundadır.

Anonim sınıflar, yukarıda da gördüğümüz gibi, bir arayüz veya soyut sınıf üzerinden oluşturulan isimsiz sınıflardır. Bu sayede, bir daha hiçbir zaman kullanmayacağımız *AdditionOperation* sınıfını yazmamıza gerek kalmaz.

Lambda ifadeleri, anonim sınıf kullanarak yazdığımız kodları daha kısa yazmamıza imkân tanır. Lambda ifadelerini kullanarak, yukarıda yazdığımız kodu aşağıdaki gibi tek satırda da yazabiliriz:

```
int result = Math.operateTwoNumbers(5, 10, (x, y) -> x + y);
System.out.println(result);
```

Kalın olarak belirttiğimiz kod bir lambda ifadesidir. Gördüğünüz gibi, anonim sınıf yerine lambda ifadesi kullandığımız zaman kodu tek satıra indirgedik.

Şimdi, lambda ifadelerini daha ayrıntılı incelemeden önce, fonksiyonel arayüzlerden bahsedelim.

Fonksiyonel arayüzler (functional interfaces)

Anonim sınıfların isimsiz sınıflar olduğunu belirtmiştik. İsimsiz bile olsalar, sınıfın yapısını belirleyebilmek için bir arayüze ihtiyaç duyarız. Lambda

ifadelerinin de isimsiz metotlar olduğunu söylemiştik; aynı şekilde, isimsiz olsa bile bir metodun yapısını belirlemek gerekir. Bunu fonksiyonel arayüzleri kullanarak yaparız.

İçinde en fazla 1 tane soyut metot bulunan arayüzlere **fonksiyonel arayüz** denir. Fonksiyonel arayüzleri lambda ifadelerini tanımlayabilmek için kullanırız. Eğer bir arayüzde birden fazla soyut metot varsa, bu arayüz ile lambda ifadesi tanımlayamayız; çünkü Java çalışma ortamı hangi metodu kullanarak lambda ifadesi yazdığımızı kestiremez.

Örneğin, aşağıdaki 3 arayüzü inceleyelim:

```
interface MyInterface1
{

    void myMethod1();

}

interface MyInterface2
{

    void myMethod2();

    default Date now()
    {
        return new Date();
    }

}

interface MyInterface3
{

    int myMethod3();

    boolean myMethod4(double myParam1, int myParam2);

}
```

MyInterface1 ve *MyInterface2* arayüzleri fonksiyonel arayüzdür; çünkü bir tane soyut metot tanımlamışlardır. *MyInterface3* ise fonksiyonel arayüz değildir; çünkü birden fazla soyut metot tanımlamıştır. Bunun anlamı şudur: *MyInterface1* ve *MyInterface2* arayüzlerini lambda ifadesi oluşturmak için kullanabiliriz.

Lambda ifadesi yazmak

Bir lambda ifadesinin yapısı aşağıdaki gibidir:

```
( [parametreler] ) ->
{
        [metodun içeriği]
}
```

Öncelikle, parantez içinde metodun parametreleri yazılır. Eğer metot parametre almıyorsa parantez boş bırakılır. Eğer metot 1 tane parametre alıyorsa, parantez yazmaya gerek yoktur; fakat 1'den fazla parametre olduğu durumlarda parantez kullanmak zorunludur.

Daha sonra lambda operatörü yazılır. Lambda operatörü sırasıyla tire (-) ve büyüktür (>) işaretlerinden oluşur (aralarında boşluk yoktur).

Daha sonra bir blok açılır ve metodun içeriği yazılır. Eğer metot tek bir satırdan oluşuyorsa blok açmaya gerek yoktur. Eğer metot void değilse (bir değer döndürüyorsa) ve tek satırdan oluşuyorsa, return deyimini kullanmaya gerek yoktur.

Şimdi, daha önce yazdığımız *Operation* arayüzü üzerinden farklı lambda ifadeleri tanımlayalım ve kullanalım:

```
final int number1 = 6;
final int number2 = 3;

Operation addition = (x, y) -> x + y;
Operation subtraction = (x, y) -> x - y;
Operation multiplication = (x, y) -> x * y;
Operation division = (x, y) -> x / y;

System.out.println(addition.operate(number1, number2));
System.out.println(subtraction.operate(number1, number2));
System.out.println(multiplication.operate(number1, number2));
System.out.println(division.operate(number1, number2));
```

Yukarıdaki kodu çalıştırdığınız zaman çıktısı aşağıdaki gibi olur:

```
9
3
18
2
```

Lambda ifadeleri ve değişkenlere erişim

Bir lambda ifadesi içinden, tanımlandığı sınıfın statik üyelerine erişilebilir. Aynı zamanda, lambda ifadelerinin **this** deyimine de erişimi vardır. Yani, this deyimini kullanarak bir lambda ifadesinde sınıfın üyelerine erişebilirsiniz.

Diğer yandan, bir metot içinde tanımlanmış değişkeni lambda ifadesi içinde kullanmak istiyorsanız, bu değişkenin sabit veya dolaylı olarak sabit olması gerekir; yani değişkenin değiştirilmemesi gerekir. Buradan da anlayabileceğimiz gibi, bu değişkenleri lambda ifadesi içinde değiştiremeyiz; çünkü böyle olsaydı değişkenin dolaylı olarak sabit olması durumu bozulurdu.

Örneğin, aşağıdaki kodu inceleyelim:

```
int number1 = 6;
int number2 = 3;

Operation operation = (x, y) ->
{
    System.out.println(number1 + number2);
    return x + y;
};

number1++;     // Değişkenin değeri değiştiriliyor, hataya sebebiyet
```

Yukarıda yazdığımız kod henüz derleme aşamasında hata alır; çünkü lambda ifadesi içinde kullandığımız *number1* değişkeninin değerini lambda ifadesi dışında değiştiriyoruz. Halbuki bu değişkenin dolaylı olarak sabit olması gerekirdi. Bu kodu aşağıdaki gibi yazdığımız zaman hata vermez:

```
final int number1 = 6;
final int number2 = 3;

Operation operation = (x, y) ->
{
    System.out.println(number1 + number2);
    return x + y;
};
```

Şimdi *number1* ve *number2* değişkenlerini sabit olarak tanımladığımız için, lambda ifadesi içinde rahatlıkla kullanabiliriz.

Metot referansı (method reference)

Metot referansı, lambda ifadeleriyle ilintili olarak dile eklenmiş bir özelliktir. Bu sayede, zaten var olan bir metodun içeriğini tekrar yazmadan, yalnızca referans vererek lambda ifadesi yerine kullanabiliriz.

Metot referansı şu şekilde verilir:

[sınıfın ismi]::[metodun ismi]

Örneğin *Operations* isminde bir sınıf oluşturalım:

```
public class Operations
{

    public static int add(int x, int y)
    {
        return x + y;
    }

    public static int subtract(int x, int y)
    {
        return x - y;
    }

    public static int multiply(int x, int y)
    {
        return x * y;
    }

    public static int divide(int x, int y)
    {
        return x / y;
    }

}
```

Gördüğünüz gibi, bu sınıfın içine yazdığımız 4 metot da *Operation* arayüzünde tanımlanan *operate()* metoduna uygundur. Bu sayede, bu metotları lambda ifadesi yerine metot referansı belirterek kullanabiliriz:

```
final int number1 = 6;
final int number2 = 3;

Operation addition = Operations::add;
Operation subtraction = Operations::subtract;
Operation multiplication = Operations::multiply;
Operation division = Operations::divide;

System.out.println(addition.operate(number1, number2));
System.out.println(subtraction.operate(number1, number2));
System.out.println(multiplication.operate(number1, number2));
System.out.println(division.operate(number1, number2));
```

Yukarıdaki kodu çalıştırdığımızda çıktısı öncekiyle aynı olur:

```
9
3
18
2
```

Öntanımlı bazı fonksiyonel arayüzler

JDK 8 ile Java diline bazı fonksiyonel arayüzler eklenmiştir. Şimdi bunlar arasından en sık kullanılan bazılarını inceleyelim:

Function<T, R> arayüzü

Yapısı aşağıdaki gibidir:

```
public interface Function<T, R>
{

    R apply(T t);

}
```

Bu arayüzün *apply()* adında bir metodu vardır. *T* türünde bir parametre alır ve *R* türünde bir değer döndürür. Aşağıdaki örneği inceleyelim:

```
Function<String, Integer> func = str -> Integer.parseInt(str) * 5;
System.out.println(func.apply("20"));
```

Bu kodu çalıştırdığınızda konsola 100 yazar.

Consumer<T> arayüzü

Yapısı aşağıdaki gibidir:

```
public interface Consumer<T>
{

    void accept(T t);

}
```

Bu arayüzün *accept()* adında bir metodu vardır. *T* türünde bir parametre alır ve bu argüman üzerinde bir işlem yapar. Aşağıdaki örneği inceleyelim:

```
StringBuilder text = new StringBuilder();
Consumer<String> append = str -> text.append(str);
append.accept("Bu ");
append.accept("bir ");
append.accept("metindir.");
System.out.println(text.toString());
```

Bu kodu çalıştırdığınızda çıktısı aşağıdaki gibi olur:

```
Bu bir metindir.
```

Supplier<T> arayüzü

Yapısı aşağıdaki gibidir:

```
public interface Supplier<T>
{

    T get();

}
```

Bu arayüzün *get()* adında bir metodu vardır. Parametre almaz ve T türünde bir değer döndürür. Aşağıdaki örneği inceleyelim:

```
final int max = 1000;
Supplier<Integer> randomNumberGenerator = () ->
{
    Random random = new Random();
    return random.nextInt(max);
};
System.out.println(randomNumberGenerator.get());
System.out.println(randomNumberGenerator.get());
System.out.println(randomNumberGenerator.get());
```

Burada 0 ile 1000 arasında rastgele sayı üreten bir *Supplier* nesnesi oluşturduk. Bu kodu çalıştırdığınız zaman konsola 0 ile 1000 arasında 3 tane rastgele sayı yazar.

Predicate<T> arayüzü

Yapısı aşağıdaki gibidir:

```
public interface Predicate<T>
{

    boolean test(T t);

}
```

Bu arayüzün *test()* adında bir metodu vardır. T türünde bir parametre alır ve boolean türünde bir değer döndürür. Aşağıdaki örneği inceleyelim:

```
Predicate<Integer> divisibleBy5 = number -> number % 5 == 0;
System.out.println(divisibleBy5.test(10));
System.out.println(divisibleBy5.test(12));
```

Burada bir sayının 5'e tam bölünüp bölünmediğini test eden bir Predicate nesnesi oluşturduk. Bu kodu çalıştırdığınız zaman çıktısı aşağıdaki gibi olur:

```
true
false
```

STREAM API

JDK 8 ile lambda ifadelerinin Java'ya eklenmesi üzerine, yine bununla ilintili olarak Stream API yazılmıştır. Basitçe söylemek gerekirse, koleksiyonlar üzerinde lambda ifadeleri kullanarak işlem yapmamızı sağlayan metotlar eklemiştir.

Stream, akış demektir. Nesnelerin art arda gelmesiyle bir akış oluşur. Akış yaratarak, bir dizi veya koleksiyonun elemanları üzerinde işlemler yapabiliriz. Akışlar, verinin nasıl depolanacağıyla ilgilenmez, yalnızca veriyi bir yerden bir yere transfer eder. Bu transfer esnasında veri üzerinde bir veya birden fazla işlem yapılması muhtemeldir. Bu işlem verinin filtrelenmesi, sıralanması veya dönüştürülmesi gibi işlemler olabilir. Bu işlem, akışın kaynağını değiştirmez; fakat yeni bir akış oluşturur. Örneğin, bir akışın içindeki nesneleri sıralarsanız, kaynak değişmez; fakat sıralı nesnelerden oluşan yeni bir akış yaratılır.

JDK 8 ile akışları, *Stream* türünde bir nesne olarak ifade edebiliriz. Stream API çok kapsamlı bir konu olsa da biz yalnızca koleksiyonlar üzerinde yapılan işlemleri inceleyeceğiz.

Bir koleksiyonun akışını elde edebilmek için, JDK 8 ile *Collection* arayüzüne *stream()* adında yeni bir metot eklenmiştir. Bu metodun yapısı aşağıdaki gibidir:

```
interface Collection<T>
{

    Stream<T> stream();

}
```

Bu metodu kullanarak bir koleksiyon için yeni bir akış oluşturabiliriz. Bu metot her çağrıldığında koleksiyon üzerinde yeni bir akış oluşturulur.

Şimdi *Stream* arayüzünün en çok kullanılan metotlarını inceleyelim. Bu metotların hepsinde aynı listeyi kullanacağız. Önce bu listeyi oluşturalım:

```
ArrayList<Integer> list = new ArrayList<>();
list.add(25);
list.add(12);
list.add(3);
list.add(89);
list.add(25);
list.add(44);
list.add(100);
list.add(7);
list.add(63);
```

forEach()

Bu metodu kullanarak akışın bütün elemanları üzerinde bir işlem yapabilirsiniz. *Consumer<T>* türünde bir parametre alır. Bu metot akışı sonlandıran bir metottur, yani bu metodu kullandıktan sonra akış üzerinde başka bir işlem yapamazsınız.

```
list
        .stream()
        .forEach(number -> System.out.println(number));
```

Gördüğünüz gibi, *forEach()* metodunu kullanarak akışın bütün elemanlarını konsola yazdırıyoruz. Bu kodu çalıştırdığınız zaman çıktısı aşağıdaki gibi olur:

```
25
12
3
89
25
44
100
7
63
```

filter()

Bu metodu kullanarak akışın elemanlarını filtreleyebilirsiniz. Predicate<T> türünde bir parametre alır. Bu teste uymayan elemanları akışa almaz.

```
list
        .stream()
        .filter(number -> number > 60)
        .forEach(number -> System.out.println(number));
```

Burada, filter() metodunu kullanarak yalnızca 60'dan büyük sayıların konsola yazdırılmasını istiyoruz. Bu kodu çalıştırdığınız zaman çıktısı aşağıdaki gibi olur:

```
89
100
63
```

distinct()

Bu metodu kullanarak akışın içinde her elemanın en fazla 1 kez yer almasını sağlayabilirsiniz. Eğer akışın içinde bir eleman daha önce tanımlanmışsa, ikinci kez yer almaz. Parametre almaz.

```
list
        .stream()
        .distinct()
        .forEach(number -> System.out.println(number));
```

Bu kodu çalıştırırsanız, listeye iki kez eklenen 25 sayısının yalnızca bir kez konsola yazdırıldığını görürsünüz:

```
25
12
3
89
44
100
7
63
```

sorted()

Bu metodu kullanarak akışın elemanlarını sıralayabilirsiniz.

```
list
        .stream()
        .sorted()
        .forEach(number -> System.out.println(number));
```

Bu kodu çalıştırdığınız zaman çıktısı aşağıdaki gibi olur:

```
3
7
12
25
25
44
63
89
100
```

Bu metodun *Comparator<T>* türünde bir parametre alan başka bir versiyonu daha vardır. Bu versiyonu kullanarak akışın sıralama algoritmasını değiştirebilirsiniz.

```
list
        .stream()
        .sorted(Comparator.reverseOrder())
        .forEach(number -> System.out.println(number));
```

Örneğin, bu kodu çalıştırırsanız, elemanların büyükten küçüğe doğru sıralanarak konsola yazdırıldığını görürsünüz:

```
100
89
63
44
25
25
12
7
3
```

limit()

Bu metodu kullanarak akış üzerinde gerçekleştireceğiniz işlemleri belli bir sayıyla sınırlandırabilirsiniz. **long** türünde bir sayıyı parametre olarak alır.

```
list
        .stream()
        .limit(5L)
        .forEach(number -> System.out.println(number));
```

Bu kodu çalıştırırsanız, yalnızca ilk 5 elemanın konsola yazdırıldığını görürsünüz:

```
25
12
3
89
25
```

skip()

Bu metodu kullanarak akışın belli sayıda elemanını atlayabilirsiniz. Bu elemanlar üzerinde işlem yapılmaz. **long** türünde bir sayıyı parametre olarak alır.

```
list
        .stream()
        .skip(5L)
        .limit(2L)
        .forEach(number -> System.out.println(number));
```

Burada, akışın ilk 5 elemanını atlıyor ve sonraki 2 elemanı konsola yazdırıyoruz:

```
44
100
```

count()

Bu metodu kullanarak akıştaki eleman sayısını öğrenebilirsiniz. Bu metot akışı sonlandıran bir metottur, yani bu metodu kullandıktan sonra akış üzerinde başka bir işlem yapamazsınız.

```
long count = list
        .stream()
        .filter(number -> number < 40)
        .distinct()
        .count();

System.out.println(count);
```

Burada, listenin içinde 40'tan küçük kaç farklı sayı olduğunu konsola yazdırıyoruz. Bu kodu çalıştırırsanız konsola 4 yazar.

anyMatch()

Predicate<T> türünde bir parametre alır ve bu testi akışın bütün elemanları üzerinde uygular. Elemanlardan herhangi biri bu testten geçiyorsa **true**, aksi halde **false** döndürür. Bu metot akışı sonlandıran bir metottur, yani bu metodu kullandıktan sonra akış üzerinde başka bir işlem yapamazsınız.

```
boolean match = list
        .stream()
        .anyMatch(number -> number < 5);

System.out.println(match);
```

Burada, listenin içinde 5'ten küçük sayı olup olmadığını test ediyoruz. Listede 5'ten küçük yalnızca 3 vardır; fakat bu bile metodun **true** döndürmesi için yeterlidir. Bu kodu çalıştırırsanız konsola **true** yazar.

allMatch()

Predicate<T> türünde bir parametre alır ve bu testi akışın bütün elemanları üzerinde uygular. Elemanların tamamı bu testten geçiyorsa **true**, aksi halde **false** döndürür. Bu metot akışı sonlandıran bir metottur, yani bu metodu kullandıktan sonra akış üzerinde başka bir işlem yapamazsınız.

```
boolean match = list
        .stream()
        .allMatch(number -> number < 5);

System.out.println(match);
```

Bu kodu çalıştırırsanız konsola **false** yazar; çünkü listede 5'ten büyük elemanlar da vardır.

noneMatch()

Predicate<T> türünde bir parametre alır ve bu testi akışın bütün elemanları üzerinde uygular. Elemanların hiçbiri bu testten geçmiyorsa **true**, aksi halde **false** döndürür. Bu metot akışı sonlandıran bir metottur, yani bu metodu kullandıktan sonra akış üzerinde başka bir işlem yapamazsınız.

```
boolean match = list
        .stream()
        .noneMatch(number -> number < 5);

System.out.println(match);
```

Bu kodu çalıştırırsanız konsola **false** yazar; çünkü listede 5'ten küçük elemanlar vardır.

map()

Akışın elemanlarını değiştirmek için bu metodu kullanabilirsiniz. *Function<T,R>* türünde bir parametre alır ve bu fonksiyonu akışın bütün elemanlarına uygular. Akışın yeni elemanları bu metottan dönen değerlerdir.

```
list
        .stream()
        .map(number -> number * 2)
        .forEach(number -> System.out.println(number));
```

Bu örnekte, akışın bütün elemanlarını 2 ile çarptık. Bu kodu çalıştırırsanız çıktısı aşağıdaki gibi olur:

```
50
24
6
178
50
88
200
14
126
```

Bu metodu kullanarak akışın içindeki elemanların türünü değiştirmek de mümkündür:

```
list
        .stream()
        .map(number -> Math.sqrt(number))
        .forEach(number -> System.out.println(number));
```

Burada akışın türünü Integer'dan Double'a değiştiriyoruz. Bu kodu çalıştırırsanız çıktısı aşağıdaki gibi olur:

```
5.0
3.4641016151377544
1.7320508075688772
9.433981132056603
5.0
6.6332495807108
10.0
2.6457513110645907
7.937253933193772
```