GİRDİ/ÇIKTI (INPUT/OUTPUT) SINIFLARI

Bütün programcıların başlangıçta öğrendiği gibi, birçok program dışarıda bulunan verilere erişmeden görevini yapamaz. Çoğu zaman programımızın ürettiği veya kullandığı verileri dış bir kaynaktan almak veya dış bir kaynağa vermek zorunda kalırız. Dış kaynaklardan gelen veriye **girdi** (*input*), dış kaynaklara yollanan veriye **çıktı** (*output*) denir.

Girdi/Çıktı (I/O) işlemlerine örnek olarak bir dosyadan veri okumak veya bir dosyaya veri yazmak verilebilir. Bu örneğimizde dış kaynak dosyadır. Aynı şekilde, ağ üzerinden gelen veriyi okumak veya ağ üzerinden veri aktarmak da bir girdi/çıktı işlemidir. Bir sunucuyla iletişime geçmek de girdi/çıktı işlemlerine örnek olarak gösterilebilir.

Java'da bu tarz işlemleri gerçekleştirebilmek için girdi/çıktı sınıflarını kullanırız. Bu sınıflar **java.io** paketi altında bulunur. Yukarıda birkaç örneğini verdiğimiz dış kaynaklar fiziksel olarak farklı olsalar da tek bir kavram altında ifade edilebilir. Bu soyut kavrama **akış** (*stream*) deriz. Bir girdi/çıktı akışı, veri üreten veya tüketen mantıksal bir birim ifade eder. Dolayısıyla, girdi/çıktı sınıfları akışlar üzerinde işlem yapmamızı sağlar. Java'nın sağladığı bu soyutlama sayesinde, fiziksel olarak farklı olsalar da bütün girdi/çıktı aygıtları üzerinde aynı şekilde işlem yapabiliriz.

Şimdi **java.io** paketi altındaki en çok kullanılan girdi/çıktı sınıflarını inceleyelim.

File sınıfı

Java'nın I/O sınıflarının çoğu akışlar üzerinde işlem yapar; fakat *File* sınıfı öyle değildir. Bu sınıfı bilgisayarımızda bulunan dosya ve klasörleri ifade etmek için kullanırız. Yani, *File* sınıfı verinin nasıl saklandığını veya okunacağını belirtmez, yalnızca dosyaların özelliklerini belirtir.

File sınıfının bazı yapılandırıcıları şu şekildedir:

File(**String** *dosyanınTamYolu*)

File(**String** *dosyanınBulunduğuDizin*, **String** *dosyanınAdı*)

File(**File** *dosyanınBulunduğuDizin*, **String** *dosyanınAdı*)

Bazı örnekleri inceleyelim:

```
File file1 = new File("C:/Windows/System32");
File file2 = new File("C:/Windows", "System32");
File dir = new File("C:/Windows/");
File file = new File(dir, "System32");
```

Yukarıdaki örnekte *C* diskinin altında bulunan *Windows* klasörünün altındaki *System32* klasörüne erişmek için 3 farklı değişken tanımladık. Görebildiğiniz gibi, aynı klasör için 3 farklı yapılandırıcı çağırabiliriz.

File sınıfının bazı önemli metotlarını aşağıdaki tabloda inceleyelim:

String getName()	Dosyanın ismini döndürür.
String getParent()	Dosyanın bulunduğu klasörün ismini *String* olarak döndürür. Eğer yoksa **null** döndürür.
File getParentFile()	Dosyanın bulunduğu klasörün ismini *File* olarak döndürür. Eğer yoksa **null** döndürür.
String getAbsolutePath()	Dosyanın tam yolunu *String* olarak döndürür.
File getAbsoluteFile()	Dosyanın tam yolunu *File* olarak döndürür.
boolean canRead()	Bu programın dosyayı okuma yetkisi olup olmadığını denetler.
boolean canWrite()	Bu programın dosyaya yazma yetkisi olup olmadığını denetler.
boolean exists()	Böyle bir dosya mevcutsa **true** döndürür.
boolean isDirectory()	Eğer bir klasör belirtiyorsa **true** döndürür.
boolean isFile()	Eğer bir dosya belirtiyorsa **true** döndürür.
long length()	Dosyanın boyutunu byte cinsinden döndürür. Eğer böyle bir dosya yoksa 0 döndürür.
boolean delete()	Dosyayı silmeye çalışır, silerse **true** döndürür.
String[] list()	Eğer bir klasör belirtiyorsa, bu klasörün içinde bulunan dosya ve klasörleri *String* dizisi olarak döndürür.
File[] listFiles()	Eğer bir klasör belirtiyorsa, bu klasörün içinde bulunan dosya ve klasörleri *File* dizisi olarak döndürür.

boolean mkdir()	Eğer bir klasör belirtiyorsa ve böyle bir klasör yoksa oluşturmaya çalışır, başarılı olursa **true** döndürür.
boolean mkdirs()	Eğer bir klasör belirtiyorsa ve böyle bir klasör yoksa oluşturmaya çalışır. Eğer dizin yolundaki klasörlerden birden fazlası yoksa hepsini oluşturmaya çalışır, başarılı olursa **true** döndürür.
boolean renameTo(File dest)	Dosyanın ismini belirtilen parametreye uygun olarak değiştirmeye çalışır, başarılı olursa **true** döndürür.

AKIŞ SINIFLARI

File sınıfını tanıttıktan sonra akış sınıflarına geçebiliriz.

Akış sınıflarını ikiye ayırabiliriz: byte akışları ve karakter akışları. Karakter akışları, dosyanın içeriğinin metin olarak anlamlı olduğu akışlara denir. Metin dosyaları buna örnek olarak verilebilir. Diğer akışlara ise byte akışları denir. Byte akışlarının içeriğini okuduğumuzda metin olarak bir anlam ifade etmezler. Görüntü dosyaları (.jpg, .png vs), PDF dosyaları byte akışlarına örnek olarak verilebilir.

Akış sınıflarının metotlarının çoğu *IOException* fırlatır. Bu hata, okuma veya yazma işlemi sırasında bir hata oluştuğunu belirtir.

Java'nın akış sınıfları 4 soyut sınıf (*abstract class*) üzerine kurulmuştur:

	Byte akışları	**Karakter akışları**
Okumak için	InputStream	Reader
Yazmak için	OutputStream	Writer

InputStream

Byte akışlarından gelen verileri okumak için yazılmış soyut bir sınıftır. Okuma işlemleri için gerekli bazı metotları tanımlamıştır. Bu metotlardan bazılarını inceleyelim:

void close()	Akışı kapatır. **Bütün akış kaynakları işlem tamamlandıktan sonra kapatılmalıdır!**
int read()	Akışta bulunan sıradaki byte değerini okur. Eğer dosyanın sonuna gelindiyse -1 döndürür.
int read(**byte**[] buffer)	Parametre olarak verilen dizinin boyutu kadar byte değerini okur ve dizinin içine atar.
byte[] readAllBytes()	Dosyanın sonuna kadar bütün byte değerlerini okur ve bir dizi halinde döndürür.
byte[] readNBytes(**int** n)	Parametre olarak verilen sayı kadar byte değeri okur ve bir dizi halinde döndürür.
long skip(**int** n)	Parametre olarak verilen sayı kadar byte değerini okumadan atlar.

OutputStream

Byte akışlarına veri yazmak için kullanılan soyut bir sınıftır. Yazma işlemleri için gerekli bazı metotları tanımlamıştır. Bu metotlardan bazılarını inceleyelim:

void close()	Akışı kapatır. **Bütün akış kaynakları işlem tamamlandıktan sonra kapatılmalıdır!**
void flush()	Eğer fiziksel olarak akışa yazılmamış byte değerleri varsa, bunların yazılması için bir sinyal gönderir.
void write(**int** c)	Akışa bir byte değeri yazar. Bu değeri parametre olarak alır.
void write(**byte**[] buffer)	Parametre olarak aldığı byte dizisinin içindeki bütün byte değerlerini sırasıyla akışa yazar.

InputStream ve *OutputStream* sınıfları soyut sınıflardır. Yani bu sınıfları tek başına kullanamayız. Ancak alt sınıfları oluşturulursa bir anlam ifade ederler. Şimdi bu sınıfların en çok kullanılan alt sınıflarını inceleyelim.

FileInputStream

Dosyaların içeriğini okumak için bu sınıfı kullanırız. Sınıfın bir örneğini alırken parametre olarak okuyacağımız dosyanın yolunu *String* veya *File* olarak veririz. Eğer okumak istediğimiz dosya mevcut değilse *FileNotFoundException* fırlatılır.

Şimdi bir dosyadan veri okumayla ilgili örnek yapalım:

```
File inputFile = new File("ornek_dosya.txt");

try
{
    FileInputStream fis = new FileInputStream(inputFile);

    int c;
    while ((c = fis.read()) != -1)
    {
        System.out.println(c);
    }

    fis.close();
}
catch (IOException ex)
{
    System.out.println("Dosyayı okurken hata meydana geldi!");
}
```

Yukarıdaki örnekte, bilgisayarımızdaki *ornek_dosya.txt* dosyasını açıyor ve içeriğindeki **byte** değerlerini konsola yazdırıyoruz. Bu kodda dikkat etmemiz gereken bazı noktaları şöyle sıralayabiliriz:

- *FileInputStream* oluştururken parametre olarak okuyacağımız dosyayı verdik.
- Okuma işlemini bir döngünün içinde yaptık; çünkü *read()* metodu yalnızca 1 byte okur.
- Dosyanın sonuna geldiğimizde hata almamamız için -1 kontrolü yaptık. Çünkü *read()* metodu dosyanın sonuna gelindiğinde -1 döndürür.
- İşlemimizi tamamladıktan sonra *close()* metodunu kullanarak akışı kapattık. **Akışlarla uğraşırken akışı kapatmayı unutmamalıyız!**
- Akış sınıflarındaki metotların çoğu *IOException* fırlatabilir. Bu yüzden yukarıdaki kodları *try-catch* bloğu içinde yazdık.

FileOutputStream

Dosyaların içeriğine yazmak için bu sınıfı kullanırız. Sınıfın bir örneğini alırken parametre olarak okuyacağımız dosyanın yolunu *String* veya *File* olarak veririz. Ayrıca ikinci parametre olarak **boolean** türünde *append* isminde bir argüman veririz. Bu argüman dosyaya ekleme yapılıp yapılmayacağını tespit etmek için kullanılır. Eğer **true** verirsek ve yazmak istediğimiz dosya mevcutsa; dosyanın içeriği korunur ve sonuna ekleme yapılır. Eğer **false** verirsek ve yazmak istediğimiz dosya mevcutsa; dosyanın mevcut içeriği silinir ve üzerine yazılır. Eğer biz bu parametreyi vermezsek varsayılan değer olarak **false** kullanılır.

```
File outputFile = new File("ornek_dosya.txt");
String text = "Bu metin dosyanın içeriğine yazılacak.";
byte[] textBytes = text.getBytes(StandardCharsets.UTF_8);

try
{
    FileOutputStream fos = new FileOutputStream(outputFile);
    fos.write(textBytes);
    fos.close();
}
catch (IOException ex)
{
    System.out.println("Dosyaya yazarken hata meydana geldi!");
}
```

Yukarıdaki örnekte şunları yaptık:

- Dosyanın içeriğine yazmak için bir metin belirledik.
- Bu metni UTF-8'e göre byte dizisine dönüştürdük.
- *FileOutputStream* kullanarak bu byte dizisini dosyaya yazdık ve daha sonra akışı kapattık.

Dosya kopyalamak

Şimdi *FileInputStream* ve *FileOutputStream* sınıflarını kullanarak bir dosya kopyalayalım:

```
File sourceFile = new File("kopyalanacak_dosya.txt");
File destinationFile = new File("yeni_dosya.txt");

try
{
    FileInputStream source = new FileInputStream(sourceFile);
    FileOutputStream destination = new FileOutputStream(destinationFile,
        false);

    int c;
    while ((c = source.read()) != -1)
    {
        destination.write(c);
    }

    destination.close();
    source.close();
}
catch (IOException ex)
{
    System.out.println("Dosyayı kopyalarken hata meydana geldi!");
}
```

BufferedInputStream

Yukarıda gördüğümüz *FileInputStream* sınıfı dosyadan okuma yapmak için kullanılıyordu. Okuma yapmak için kullandığımız *read()* metodu her seferinde disk üzerindeki dosyaya gidiyor ve fiziksel olarak dosyanın içinde bulunan 1 byte değerini okuyordu.

Sabit diske erişim RAM'e erişime göre daha yavaştır. Sıklıkla sabit disk üzerinde işlem yapmak programımızın hızını nispeten yavaşlatır. Peki bunun önüne nasıl geçebiliriz? Şöyle bir çözüm düşünülebilir: dosya üzerinde her seferinde tek bir byte değeri okumaktansa birden fazla byte değerini okuyabilir ve bu değerleri bir dizi halinde RAM'de tutabiliriz. Böylece sık sık sabit diske erişmek yerine RAM'e erişir ve işlemlerimizi daha hızlı yapabiliriz.

BufferedInputStream sınıfı bize bu işlevselliği sağlar. Parametre olarak bir *InputStream* nesnesi ve bir dizi boyutu alır. Siz o *InputStream* nesnesi üzerindeki *read()* metodunu çağırdığınızda, tek bir byte değeri okumak yerine verdiğiniz dizi boyutu kadar okuma yapar ve bunu hafızada tutar. Dizinin tamamını okuduğunuzda tekrar dosyaya başvurur ve dizi boyutu kadar yeni

bir okuma yapar. Yani, gerçek akış kaynağına erişimi olabildiğince azaltarak RAM üzerinden okuma işlemi yapar.

Şimdi *BufferedInputStream* sınıfının işlevini anlayabilmek için iki örnek yapalım:

```
File inputFile = new File("ornek_dosya.docx");

try
{
    long start = System.currentTimeMillis();
    FileInputStream fis = new FileInputStream(inputFile);

    int c;
    while ((c = fis.read()) != -1)
    {
        System.out.println(c);
    }

    fis.close();
    long end = System.currentTimeMillis();

    System.out.println("İşlem " + (end - start) + " milisaniye sürdü.");
}
catch (IOException ex)
{
    System.out.println("Dosyayı okurken hata meydana geldi!");
}
```

Yukarıdaki örnekte *BufferedInputStream* kullanmadan, *FileInputStream* ile dosyadan okuma yapıyoruz. Okuma işleminin başlangıcında ve sonunda tarihe bakıyor ve işlemin kaç milisaniye sürdüğünü tespit ediyoruz. Bu kodu 37 KB boyutundaki bir .docx dosyası üzerinde çalıştırdığım zaman 177 milisaniye sürdü. Tabi bu değerin işlemci hızı vs. gibi ortam faktörlerine göre değişebileceğini unutmayın. Şimdi aynı örneği *BufferedInputStream* kullanarak tekrar yapalım:

```
File inputFile = new File("ornek_dosya.docx");

try
{
    long start = System.currentTimeMillis();
    FileInputStream fis = new FileInputStream(inputFile);
    BufferedInputStream bis = new BufferedInputStream(fis);

    int c;
    while ((c = bis.read()) != -1)
    {
        System.out.println(c);
    }

    fis.close();
    long end = System.currentTimeMillis();

    System.out.println("İşlem " + (end - start) + " milisaniye sürdü.");
}
catch (IOException ex)
{
    System.out.println("Dosyayı okurken hata meydana geldi!");
}
```

Yukarıdaki kodu aynı dosya üzerinde çalıştırdığım zaman 120 milisaniye sürdü. Arada 57 milisaniyelik fark olduğuna dikkatinizi çekerim. Bu fark *BufferedInputStream* sınıfının verileri hafızaya atmasından kaynaklanmaktadır.

BufferedInputStream nesnesi oluştururken parametre olarak bir dizi boyutu verebilirsiniz. Eğer vermezseniz varsayılan olarak bu değer 8192 olur. Yani *BufferedInputStream* nesnesi akışlar üzerinde varsayılan olarak **8 KB** büyüklüğünde okumalar yapar.

BufferedOutputStream

Akışlara veri yazmak için kullanılır. *BufferedInputStream* sınıfına benzer şekilde çalışır. Amacı fiziksel akışa erişimi olabildiğince azaltmaktır. Bunun için hafızada bir dizi oluşturur ve değerleri bu diziye yazar. Dizi tamamen dolduğunda dizinin içindeki verileri gerçek akışa yazar. *BufferedOutputStream* kullanırken dikkat etmeniz gereken nokta şudur: yazılacak verilerin sonuna gelindiğinde veriler gerçek akışa yazılmamış

olabilir. Bu yüzden, yazma işleminin tamamlanması için *flush()* metodunu kullanmalısınız.

ByteArrayInputStream

Bir byte dizisini tıpkı bir akış gibi okumanızı sağlar. Bu sınıfın sağladığı faydayı bir örnekle anlatalım. Örneğin, dosyalar üzerinde okuma yapmak için bir kod yazdınız. Fakat daha sonra programınız gelişti ve internet üzerinden veri okur hale geldiniz. Bu veriler size byte dizisi halinde geliyor ve siz bu akışı okumak için yeni bir kod yazmak zorundasınız. Bu sınıfı kullanarak yeni bir kod yazmaktansa, byte dizisini bir akış olarak değerlendirebilir ve aynı kodu kullanabilirsiniz.

ByteArrayOutputStream

Hedef olarak bir byte dizisi kullanan akış sınıfıdır. Bu sınıfı kullanarak akış üzerinde yazdığınız veriler nihai olarak size bir byte dizisi olarak sunulur.

ObjectInputStream ve ObjectOutputStream sınıfları

Java nesneleri hafızada tutulurlar. Bazen bir nesnenin hafızadaki **anlık görüntüsünü** (*snapshot*) daha sonra tekrar kullanmak üzere kaydetmek isteyebilirsiniz. Örneğin, birden fazla sunucunuz var ve programınız bu sunucular üzerinde dağıtık olarak çalışıyor. Bir sunucu üzerinde bulunan bir nesne diğer bir sunucu üzerinde bulunmuyor olabilir. Bu nesnenin sunucular arasında aktarılması ihtiyacı doğabilir.

Nesnelerin hafızadaki anlık durumu bir byte dosyası olarak kaydedilebilir. Bu işleme **serialization** denir. Daha sonra bu dosya okunup nesne tekrar hafızaya alınabilir ve kullanılabilir. Bu işleme **deserialization** denir.

Nesneleri serialize etmek için *ObjectInputStream*, deserialize etmek için *ObjectOutputStream* sınıfı kullanılır.

Şunu da önemle belirtmek gerekir ki, bir nesneyi serialize edebilmek için o sınıfın *Serializable* arayüzünü uygulaması gerekir.

Reader

Karakter akışlarından gelen verileri okumak için yazılmış soyut bir sınıftır. Okuma işlemleri için gerekli bazı metotları tanımlamıştır. Bu metotlardan bazılarını inceleyelim:

void close()	Akışı kapatır. **Bütün akış kaynakları işlem tamamlandıktan sonra kapatılmalıdır!**
int read()	Akışta bulunan sıradaki karakteri okur. Eğer dosyanın sonuna gelindiyse -1 döndürür.
int read(**char**[] buffer)	Parametre olarak verilen dizinin boyutu kadar karakteri okur ve dizinin içine atar. Bu dizinin türünün *InputStream* sınıfındakinden farklı olarak **char** olduğuna dikkatinizi çekerim.
boolean ready()	Akışın okunmaya hazır olup olmadığını denetler.
long skip(**int** n)	Parametre olarak verilen sayı kadar karakteri okumadan atlar.

Writer

Karakter akışlarına veri yazmak için kullanılan soyut bir sınıftır. Yazma işlemleri için gerekli bazı metotları tanımlamıştır. Bu metotlardan bazılarını inceleyelim:

void close()	Akışı kapatır. **Bütün akış kaynakları işlem tamamlandıktan sonra kapatılmalıdır!**
void flush()	Eğer fiziksel olarak akışa yazılmamış karakterler varsa, bunların yazılması için bir sinyal gönderir.
void write(**int** c)	Akışa bir karakter yazar. Bu değeri parametre olarak alır.
void write(**char**[] buffer)	Parametre olarak aldığı karakter dizisinin içindeki bütün karakterleri sırasıyla akışa yazar. Bu dizinin türünün *OutputStream* sınıfındakinden farklı olarak **char** olduğuna dikkatinizi çekerim.

void write(**String** s)	Parametre olarak aldığı metni akışa yazar.
Writer append(**char** c)	Parametre olarak aldığı karakteri akışın sonuna ekler, daha sonra kendini döndürür. Zincirleme metotlar yazabilmek amacıyla eklenmiştir.

FileReader

Metin dosyalarının içeriğini okumak için bu sınıfı kullanırız. Sınıfın bir örneğini alırken parametre olarak okuyacağımız dosyanın yolunu *String* veya *File* olarak veririz. Eğer okumak istediğimiz dosya mevcut değilse *FileNotFoundException* fırlatılır.

Şimdi *FileInputStream* başlığında yaptığımız örneği *FileReader* kullanarak tekrar yapalım:

```
File inputFile = new File("ornek_dosya.txt");

try
{
    FileReader fileReader = new FileReader(inputFile);

    int c;
    while ((c = fileReader.read()) != -1)
    {
        System.out.print((char) c);
    }

    fileReader.close();
}
catch (IOException ex)
{
    System.out.println("Dosyayı okurken hata meydana geldi!");
}
```

Bu örnekte *FileInputStream* yerine *FileReader* kullandık. Ayrıca read() metoduyla okuduğumuz **int** türündeki karakteri **char** türüne dönüştürdüğümüze dikkat edin.

FileWriter

Metin dosyalarının içeriğine yazmak için bu sınıfı kullanırız. Sınıfın bir örneğini alırken parametre olarak okuyacağımız dosyanın yolunu *String* veya *File* olarak veririz. Ayrıca ikinci parametre olarak **boolean** türünde *append* isminde bir argüman veririz. Bu argüman dosyaya ekleme yapılıp yapılmayacağını tespit etmek için kullanılır. Eğer **true** verirsek ve yazmak istediğimiz dosya mevcutsa; dosyanın içeriği korunur ve sonuna ekleme yapılır. Eğer **false** verirsek ve yazmak istediğimiz dosya mevcutsa; dosyanın mevcut içeriği silinir ve üzerine yazılır. Eğer biz bu parametreyi vermezsek varsayılan değer olarak **false** kullanılır.

BufferedReader ve BufferedWriter sınıfları

BufferedInputStream ve *BufferedOutputStream* sınıflarının karakter akışları için karşılıklarıdır.

CharArrayReader ve CharArrayWriter sınıfları

Bir karakter dizisini tıpkı bir akış gibi okumak için *CharArrayReader* sınıfını kullanırız.

Bir karakter dizisine tıpkı bir akış gibi yazmak için ise *CharArrayWriter* sınıfını kullanırız.

InputStreamReader

InputStream türündeki bir akış kaynağını *Reader* türüne dönüştürmek için bu sınıf kullanılır. Eğer parametre olarak aldığınız *InputStream* türündeki bir akış kaynağının karakter akışı olduğunu biliyor ve *Reader* sınıfının işlevselliğinden faydalanmak istiyorsanız bu sınıfı kullanmalısınız:

```
try
{
    InputStream inputStream = new FileInputStream("ornek_dosya.txt");
    Reader reader = new InputStreamReader(inputStream);
}
catch (IOException ex)
{
    System.out.println("Bir hata meydana geldi!");
}
```

OutputStreamWriter

OutputStream türündeki bir akış kaynağını *Writer* türüne dönüştürmek için bu sınıf kullanılır. Eğer parametre olarak aldığınız *OutputStream* türündeki bir akış kaynağının karakter akışı olduğunu biliyor ve *Writer* sınıfının işlevselliğinden faydalanmak istiyorsanız bu sınıfı kullanmalısınız:

```
try
{
    OutputStream outputStream = new FileOutputStream("ornek_dosya.txt");
    Writer writer = new OutputStreamWriter(outputStream);
}
catch (IOException ex)
{
    System.out.println("Bir hata meydana geldi!");
}
```

DÜZENLİ İFADELER (REGEX)

Bazen bir metnin belli bir desene uygun olup olmadığını denetlemek isteriz. Örneğin, elinizde bir metin var ve siz bu metnin bir eposta adresi olup olmadığını test etmek istiyorsunuz. Bu gibi durumlarda çok fazla kontrol kodu yazmak gerekir. Fakat bunun önüne geçebilmek için **düzenli ifadeler** (***regex*** – *regular expressions*) kavramı ortaya atılmıştır.

Regex, yalnızca Java'ya özgü olmayıp bütün programlama dillerinde mevcuttur. Dolayısıyla regex kuralları herhangi bir dile ait değildir. Her programlama dili için regex motorları yazılmıştır.

Regex kullanarak bir desen oluştururuz ve bir metnin bu desene uyup uymadığını kontrol ederiz. Java'da bunu ***Pattern*** ve ***Matcher*** sınıflarını kullanarak yaparız.

Pattern sınıfı

Bir regex desenini belirtmek için *Pattern* sınıfını kullanırız. Bu sınıfın yapılandırıcıları bizden gizlenmiştir. Dolayısıyla *Pattern* nesnesi oluşturmak için **statik** *compile()* metodunu kullanırız. Bu metodun yapısı aşağıdaki gibidir:

public static Pattern compile(**String** pattern)

Parametre olarak *String* türünde bir regex deseni veririz ve bu deseni temsil eden bir *Pattern* nesnesi oluşturmuş oluruz.

Matcher sınıfı

Matcher sınıfının yapılandırıcılarına erişemeyiz. Bu sınıfı ancak bir *Pattern* nesnesi üzerinden oluşturabiliriz. Bunun için *Pattern* sınıfının *matcher()* metodunu kullanırız. Bu metodun yapısı şu şekildedir:

public Matcher matcher(**CharSequence** str)

Bir *Pattern* nesnesi oluşturduktan sonra bu metodu kullanarak bir *Matcher* nesnesi elde ederiz. Parametre olarak bir metin veririz ve bu *Matcher* nesnesini kullanarak metnin o desene uyup uymadığını denetleriz.

Matcher sınıfının en çok kullanılan metodu şudur:

public boolean matches()

Bu metot, metnin desene uyup uymadığı bilgisini döndürür. Eğer parametre olarak verilen metin, *Pattern* nesnesiyle belirtilen regex desenine uyuyorsa true döndürür. Eğer metnin içinde desene uyan birden fazla kısım mevcutsa bunu *find()* metoduyla tespit ederiz:

public boolean find()

Bu metot, metnin içinde desene uyan bir kısım olduğu sürece **true**, aksi halde **false** döndürür.

Regex deseni tanımlamak

Regex desenlerinin kendine özgü bir dili vardır. En çok kullanılan bazı kuralları burada anlatarak bazı regex desenlerini tanımlayalım.

Harf ve rakamlar

Regex deseni içinde kullanılan harf ve rakamlar metindeki harf ve rakamlarla eşleşir.

Joker karakteri

Nokta işaretinin (.) regex deseni içinde özel bir anlamı vardır. Bu karakter, metnin içindeki herhangi bir karakterle eşleşir.

Desen:	ka.ak
kavak	EŞLEŞİR
kazak	EŞLEŞİR
kaçak	EŞLEŞİR
kayak	EŞLEŞİR
kaymak	EŞLEŞMEZ

Escape karakteri

Regex deseni içinde özel bir anlamı olan karakterleri desen içinde kullanmak istiyorsanız önüne escape karakteri (\) koymanız gerekir. Örneğin, metnin içinde nokta aramak istiyorsanız desen içinde \\. kullanmanız gerekir.

Desen:	\\.
abc.def	EŞLEŞİR
abcdef	EŞLEŞMEZ

Tekrar belirteçleri

Metnin içinde bir karakterin birden fazla kez tekrarlanacağı durumlarda tekrar belirteçlerini kullanırız. 6 farklı tekrar belirteci vardır:

?	Kendisinden önceki karakterin metinde bulunmadığını veya en fazla 1 defa bulunduğunu belirtir.
*****	Kendisinden önceki karakterin metinde bulunmadığını veya 1 ya da daha fazla defa bulunduğunu belirtir.
+	Kendisinden önceki karakterin metinde en az 1 defa bulunduğunu belirtir.
{n}	Kendisinden önceki karakterin metinde *n* defa bulunduğunu belirtir.
{n,}	Kendisinden önceki karakterin metinde en az *n* defa bulunduğunu belirtir.
{n,m}	Kendisinden önceki karakterin metinde en az *n*, en çok *m* defa bulunduğunu belirtir.

Desen:	sa?t
st	EŞLEŞİR
sat	EŞLEŞİR
saat	EŞLEŞMEZ

Desen:	sa+t
st	EŞLEŞMEZ
sat	EŞLEŞİR
saat	EŞLEŞİR

Desen:	sa*t
st	EŞLEŞİR
sat	EŞLEŞİR
saat	EŞLEŞİR

Desen:	sa{2}t
st	EŞLEŞMEZ
sat	EŞLEŞMEZ

saat	EŞLEŞİR

Desen:	sa{2,}t
sat	EŞLEŞMEZ
saat	EŞLEŞİR
saaat	EŞLEŞİR

Desen:	sa{2,4}t
sat	EŞLEŞMEZ
saat	EŞLEŞİR
saaat	EŞLEŞİR
saaaat	EŞLEŞİR
saaaaat	EŞLEŞMEZ

Karakter kümesi

Metnin içinde birden fazla karakterle eşleşebilecek bir kısım varsa bu karakterleri belirtmek için köşeli parantezler kullanılır. Köşeli parantezler tek bir karakter belirtir; fakat bu karakter içinde belirtilen karakterlerden herhangi biri olabilir.

Desen:	ka[yzm]ak
kavak	EŞLEŞMEZ
kazak	EŞLEŞİR
kaçak	EŞLEŞMEZ
kayak	EŞLEŞİR
kaymak	EŞLEŞMEZ

Karakter kümesinin sonuna tekrar belirteçleri koyulabilir, böylece birden fazla kez tekrarlanması sağlanır.

Desen:	ka[yzm]+ak
kazak	EŞLEŞİR
kayak	EŞLEŞİR
kaymak	EŞLEŞİR

Dışlama karakteri

Yukarıda bir karakter kümesi belirlemeyi görmüştük. Eğer karakter kümesinin içindeki karakterlerle değil, o karakterlerin dışındaki bütün karakterlerle eşlenmesi isteniyorsa, dışlama karakteri (^) kullanılır. Dışlama karakterleri o karakterlerin eşleşmemesini sağlar. Eper dışlama karakteri kullanılacaksa, küme içine yazılacak ilk karakter olmalıdır.

Desen:	ka[^yzm]ak
kavak	EŞLEŞİR
kazak	EŞLEŞMEZ
kaçak	EŞLEŞİR
kayak	EŞLEŞMEZ

Aralık karakteri

Belli bir aralığın içindeki karakterlerin tümünü küme içinde yazmak yerine aralık karakteri (-) kullanılabilir.

Desen:	[0-3]
0	EŞLEŞİR
1	EŞLEŞİR
2	EŞLEŞİR
3	EŞLEŞİR
4	EŞLEŞMEZ

Desen:	[a-c]
a	EŞLEŞİR
b	EŞLEŞİR
c	EŞLEŞİR
d	EŞLEŞMEZ
e	EŞLEŞMEZ

Bütün harflerle eşleşmek için **[a-zA-Z]**; bütün rakamlarla eşleşmek için **[0-9]** desenlerini kullanabilirsiniz.

Gruplama karakteri

Gruplamak istediğiniz karakterleri parantez içine alabilirsiniz. Bu sayede, örneğin, bir karakter grubunun birden fazla kez tekrarını sağlayabilirsiniz.

Desen:	(ana)+
an	EŞLEŞMEZ
ana	EŞLEŞİR
anaana	EŞLEŞİR

Bir regex örneği

Regex deseni tanımlarken sık kullanılan bazı kuralları yukarıda anlattık. Fakat regex konusu bundan çok daha kapsamlıdır. Daha ayrıntılı bilgi için araştırma yapmanızı şiddetle tavsiye ederim. Şimdi biz regex kullanan bir Java örneği yazalım. Bu örnekte bir metnin "0(5XX) XXX-XX-XX" desenine uyup uymadığını kontrol edeceğiz.

```
Pattern pattern = Pattern.compile("0\\(5[0-9]{2}\\) [0-9]{3}-[0-9]{2}-
      [0-9]{2}");

String[] numbers = {
        "0(532) 315-23-79",
        "0(554) 289-213-124",
        "0531981-66-34",
        "0(567) 144-78-63",
};

for (String number : numbers)
{
    Matcher matcher = pattern.matcher(number);

    if (matcher.find())
    {
        System.out.println(number + " numarası desene uyuyor.");
    }
    else
    {
        System.out.println(number + " numarası desene UYMUYOR.");
    }
}
```

Yukarıdaki kodu çalıştırdığınızda çıktısı aşağıdaki gibi olur:

```
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
```