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.