کارگاه کار با فایل

درس برنامه‌سازی پیشرفته و کارگاه

استاد درس: دکتر مهدی قطعی

استاد کارگاه: بهنام یوسفی مهر

مقدمه

تا به اینجا با انواع مختلفی از داده‌ها مثل `bool`، `int`، `char` و ساختمان داده‌های پیچیده‌تر مانند `String` کار کرده‌اید؛ داده‌ها را به عنوان ورودی دریافت کرده، پردازش کرده و در نهایت خروجی داده‌اید. با این وجود تمامی این داده‌ها را در یک اجرای برنامه نیاز داشته‌اید و هیچ گاه از این داده‌ها با این ذهنیت استفاده نکرده‌اید که در آینده هم به آن‌ها نیاز خواهید داشت. در واقع حتی اگر با ساختار و نحوه‌ی کار کامپیوتر هم آشنا نباشید، می‌دانید که پس از پایان اجرای برنامه دیگر به داده‌های برنامه دسترسی نخواهید داشت (یا حداقل دسترسی به این داده‌ها سخت‌تر خواهد بود!) و پس از قطع برق، این داده‌ها به کلی توسط کامپیوتر فراموش می‌شوند. به همین خاطر است که این حافظه‌ی داخل کامپیوتر را فرار[1] می‌نامند. از این رو به هنگام نوشتن این جملات برای شما، به دلیل ترسی که از قطع برق و از بین رفتن متن داریم، پس از نوشتن یک یا چند جمله آن را ذخیره می‌کنیم!

در طرف مقابل حافظه‌ی فرار که برای نگهداری داده‌ها و برنامه‌ها در هنگام اجرا استفاده می‌شود و آن را حافظه‌ی اولیه (اصلی)[2] می‌نامند، حافظه‌ی غیرفرار[3] قرار دارد که داده‌ها و برنامه‌ها را در بین اجرای برنامه‌ها ذخیره می‌کند و حافظه‌ی ثانویه[4] نامیده می‌شود. دسترسی به حافظه‌ی اصلی به دلیل ارتباط مستقیم با CPU سریعتر از حافظه‌ی ثانویه است.

اگر بخواهیم که داده‌ها و اطلاعات برنامه روی حافظه‌ی ثانویه ذخیره شوند تا نه فقط در هنگام اجرای برنامه، بلکه در بین اجراهای مختلف برنامه (یعنی بازه‌های پس از پایان برنامه و اجرای مجدد آن) قابل دسترسی باشند، باید از APIهای مخصوصی استفاده کنیم که توسط I/O (`Input/Output`) به ما دسترسی نوشتن اطلاعات، ویرایش، ذخیره و حذف آنها را می‌دهند.

نوشتن داده‌های متنی

فرض کنید می‌خواهیم برنامه‌ای بنویسیم که اطلاعات دانشجوها را دریافت کرده و در یک فایل متنی با نام `student.txt` بنویسد.

قبل از هر چیز بهتر است ساختاری داشته باشیم که بتوانیم در هنگام اجرای برنامه اطلاعات ورودی را پردازش کرده و آنگاه در فایل خروجی ذخیره کنیم. پس در ادامه کلاس `Employee` را تعریف می‌کنیم.

نکته: یکی از مفاهیم مهم در برنامه‌نویسی شئ‌گرا کپسوله‌سازی[5] است؛ یعنی می‌خواهیم اطلاعات و داده‌های تعریف شده داخل کلاس را از تغییرات خطرناک خارجی حفظ کنیم. اما برای این کار چه باید بکنیم؟ کافی است برای شروع متغیرهای کلاس را `private`

---

[1] volatile
[2] primary (main) memory
[3] nonvolatile
[4] secondary memory
[5] encapsulation

تعریف کرده و برای دسترسی به آنها متدهای public مخصوصی تعریف کنیم که به طور غیرمستقیم دسترسی به این متغیرها را فراهم می‌کنند؛ مثل متدهای getFirstName()، getFamilyName()... که در اینجا تعریف کرده‌ایم.

```
public class Student
{
    private String firstName;
    private String familyName;
    private int age;
    private int entryYear; // four-digit number
    private double gpa;    // on scale of 4

    public Student(String firstName, String familyName, int age, int entryYear, double gpa)
    {
        this.firstName = firstName;
        this.familyName = familyName;
        this.age = age;
        this.entryYear = entryYear;
        this.gpa = gpa;
    }

    public String getFirstName()
    {
        return firstName;
    }

    public String getFamilyName()
    {
        return familyName;
    }

    public int getAge()
    {
        return age;
    }

    public int getEntryYear()
    {
        return entryYear;
    }

    public double getGpa()
    {
        return gpa;
    }
}
```

همان طور که مشخص است اطلاعات دانشجو که برای ما اهمیت دارند شامل نام، نام خانوادگی، سن، سال ورود و نمره‌ی GPA او می‌باشند.

در ادامه برنامه‌ی ساده‌ای می‌نویسیم که اطلاعات دانشجوی فرضی زیر را در یک فایل با نام student.txt می‌نویسد.

نام: Ryan

نام خانوادگی: Smith

سن: 19

سال ورود: 1402

GPA: 3.7

تمرین ۱: آیا می‌توانید کلاس فوق را به گونه‌ای تغییر دهید که با هر بار ایجاد یک نمونه از Student، پیامی مبنی بر ایجاد نمونه چاپ شود؟ مثلاً پس از ایجاد دانشجوی فرضی فوق، پیام زیر چاپ شود:

```
19 year old student Ryan Smith who entered university in 1402 and has a current GPA of 3.7 has just been created!
```

برای اینکه از API مربوط به I/O در جاوا استفاده کنیم به پکیج java.io نیاز داریم. می‌توانید با مراجعه با اسناد رسمی جاوا اطلاعات بیشتری درمورد این پکیج به دست آورید.

اما توجه کنید که کدهای مربوط به I/O خطرناک هستند و ممکن است در زمان اجرای برنامه خطا دهند! مثلاً ممکن است فایلی که به آن نیاز داریم به هر دلیلی بارگذاری نشود! آن وقت چه باید کرد؟!

یکی از راه‌هایی که به کامپایلر (و همچنین IntelliJ که خیلی هوشمندانه از ما ایراد می‌گیرد!) می‌فهمانیم که حواسمان هست به اینکه ممکن است در هنگام اجرا مشکلی به وجود بیاید، استفاده از try/catch است. try/catch به کامپایلر می‌گوید که ما آماده‌ی مدیریت مشکل احتمالی هستیم.

پس در بلوک مربوط به try کد خطری و در بلوک catch آنچه قرار است در هنگام بروز خطر انجام دهیم را قرار می‌دهیم.

به متغیر ex در داخل catch توجه کنید؛ این همان خطری است که توسط کد خطری انداخته می‌شود و ما همان طور که آرگومان‌های یک متد را تعریف می‌کنیم، در اینجا هم متغیر احتمالی ex را تعریف کرده‌ایم؛ شیئی از کلاس IOException. به جای IOException می‌توانستیم Exception هم بنویسیم که در واقع والد همه‌ی خطرهاست! (اگر این جمله برایتان مفهومی ندارد، نگران نباشید! در ادامه‌ی درس که به مفهوم ارث‌بری رسیدید می‌توانید به این قسمت برگشته و متوجه شوید!)

```
import java.io.*;

public class Main
{
    public static void main(String[] args)
    {
        try
        {
            Student student = new Student("Ryan", "Smith", 19, 1402, 3.7);

            FileWriter writer = new FileWriter("student.txt");

            writer.write(student.getFirstName() + ',' + student.getFamilyName() + ',' +
student.getAge() + ',' + student.getEntryYear() + ',' + student.getGpa());

            writer.close();
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

خط زیر اعضای عمومی پکیج `java.io` از جمله کلاس `FileWriter` را ایمپورت می‌کند.

```
import java.io.*;
```

در این خط اگر فایل `student.txt` وجود نداشته باشد ایجاد می‌شود.

```
FileWriter writer = new FileWriter("student.txt");
```

در خط فوق رشته‌ای که می‌خواهیم در فایل `student.txt` نوشته شود را از طریق متد `write()` به `writer` می‌دهیم.

`FileWriter` هرچیزی را که شما به آن بدهید بی‌درنگ داخل فایل مورد نظر می‌نویسد. در زیر رشته‌ای که می‌خواهیم داخل فایل نوشته شود را به `writer` داده‌ایم.

```
writer.write(student.getFirstName() + ',' + student.getFamilyName() + ',' + student.getAge() +
',' + student.getEntryYear() + ',' + student.getGpa());
```

تمرین ۲: با توجه به آنچه که درمورد حافظه‌ی اولیه و ثانویه می‌دانید بگویید که چرا استفاده‌ی مکرر از متد `write()` روی `FileWriter` به صرفه نیست؟ (راهنمایی: فایل متنی در کجا ذخیره شده است؟!)

در نهایت باید `writer` را با فراخوانی متد `close()` ببندیم. این متد به جریان I/O پایان می‌دهد.

```
writer.close();
```

پس از پایان اجرای برنامه فایل `student.txt` حاوی متن زیر خواهد بود:

```
Ryan,Smith,19,1402,3.7
```

تمرین ۳: `writer.close()` را حذف کنید و برنامه را دوباره اجرا کنید. فایل `student.txt` را بررسی کرده و در صورت تفاوت، علت آن را ذکر کنید.

تمرین ٤: پس از آنکه از اجرای درست برنامه مطمئن شدید آن را مجدداً اجرا کنید. انتظار دارید داخل فایل متنی چه چیزی نوشته شده باشد؟ فایل متنی را بررسی کنید. آیا مشاهدات شما همان طور است که انتظار داشتید؟

استفاده از بافر[6] برای نوشتن داده‌های متنی

روش بهینه‌تر برای نوشتن داده‌های متنی استفاده از بافر است. بافر را می‌توان مثل سطلی در نظر گرفت که در آن هر آنچه که می‌خواهیم داخل فایل بنویسیم موقتاً ذخیره می‌شود و تنها زمانی که سطل پر شد، هر آنچه که داخل آن است به فایل مقصد منتقل می‌شود. البته می‌توان قبل از پر شدن بافر (یا همان سطل فرضی!) محتویاتش را به فایل مقصد انتقال داد. متدی که می‌توان برای این منظور استفاده کرد `flush()` است.

در ادامه اطلاعات همان دانشجو را با استفاده از `BufferedWriter` می‌نویسیم.

```
import java.io.*;

public class Main
{
    public static void main(String[] args)
    {
        try
        {
            Student student = new Student("Ryan", "Smith", 19, 1402, 3.7);
            FileWriter writer = new FileWriter("student.txt");
            BufferedWriter bufferedWriter = new BufferedWriter(writer);
            bufferedWriter.write(student.getFirstName() + ',' + student.getFamilyName() + ','
+ student.getAge() + ',' + student.getEntryYear() + ',' + student.getGpa());

            bufferedWriter.close();
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

[6] buffer

توجه کنید که تعریف متغیر writer لازم نیست و می‌توان bufferedWriter را به این صورت تعریف کرد:

```
BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter("student.txt"));
```

نکته: در قطعه کد بالا فراخوانی close() فقط روی bufferedWriter کافی است و جریان writer هم بسته می‌شود.

کلاس java.io.File

کلاس java.io.File یکی از کلاس‌های قدیمی در API جاواست که جایگزین‌های مدرنی برای آن وجود دارد، ولی با این حال احتمالاً در برنامه‌نویسی جاوا با کدهایی که از این کلاس استفاده می‌کنند برخورد خواهید کرد. برای کدهای نوین‌تر استفاده از java.nio.File به جای این کلاس پیشنهاد می‌شود.

کلاس java.io.File نماینده‌ی فایلی روی دیسک است، اما نه محتویات آن. این کلاس را به مثابه بسته‌ای شامل اطلاعات (نه محتویات!) و نشانی فایل در نظر بگیرید، و نه خود آن؛ پس در اکثر متدهایی که نام فایل را به صورت رشته در کانستراکتور خود می‌گیرند می‌توانید به جای رشته‌ی نام فایل، شیئی از این کلاس را به آنها بدهید. با این حال این کلاس متدهای خواندن و نوشتن ندارد. با استفاده از این کلاس می‌توانید مطمئن شوید که آیا فایل مورد نظر وجود دارد یا خیر و در صورتی که خطایی دریافت نکردید و مطمئن شدید که فایل مورد نظر بدون خطا خوانده شده، ادامه‌ی کارها را توسط متدهای دیگر انجام دهید.

تمرین ۵: در کدهای قبلی یک شئ File ساخته و آن را به جای رشته‌ی نام فایل یعنی student.txt به FileWriter بدهید.

تمرین ۶: سه دانشجو از کلاس Student ایجاد کرده و اطلاعات هر کدام را روی سه خط متوالی داخل فایلی با نام students.txt بنویسید.

خواندن داده‌های متنی

حال در این قسمت می‌خواهیم اطلاعات دانشجویی را که در قسمت قبلی داخل student.txt نوشتیم، بخوانیم. برای این کار از یک شئ File استفاده می‌کنیم که نماینده‌ی فایل است، از یک شئ از کلاس FileReader استفاده می‌کنیم که خواندن فایل را برای ما انجام می‌دهد و در نهایت از بافری که BufferedReader در اختیارمان قرار می‌دهد برای بهینه‌تر کردن خواندن بهره می‌بریم.

```java
import java.io.*;

public class Main
{
    public static void main(String[] args)
    {
        try
        {
            File myFile = new File("student.txt");
            FileReader fileReader = new FileReader(myFile);
            BufferedReader reader = new BufferedReader(fileReader);

            String line;
            while ((line = reader.readLine()) != null)
                System.out.println(line);

            reader.close();
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

متن را خط به خط می‌خوانیم؛ به همین دلیل است که در ابتدا رشته‌ی `line` را تعریف کرده و از متد `readLine()` استفاده می‌کنیم. به عبارت `((line = reader.readLine()) != null)` توجه کنید؛ در جاوا خروجی عملگر انتسابی `=` همان مقدار انتسابی است. بنابراین در اینجا هم `reader.readLine()` اجرا، خروجی آن در `line` ذخیره شده و در ادامه با `null` مقایسه می‌شود.

در واقع تا زمانی که خروجی `reader.readLine()` مخالف `null` است در یک حلقه هر خط چاپ می‌شود.

حال فرض کنید می‌خواهیم کلاس مربوط به دانشجوی فرضی که اطلاعاتش در فایل متنی `student.txt` ذخیره شده را از نو بسازیم و اطلاعات آن را به طور منظم و منسجمی در خروجی چاپ کنیم. توجه کنید که برای این کار از قبل می‌دانیم اطلاعات دانشجو به صورت زیر در `student.txt` ذخیره شده است:

```
<firstName>,<familyName>,<age>,<entryYear>,<gpa>
```

پس کافی است پس از خواندن خطی که در `student.txt` نوشته شده، رشته‌ها و اعداد بین ویرگول‌ها را جدا کرده و آنها را به ترتیب به کانستراکتور `Student` بدهیم تا شئ جدیدی از این کلاس ایجاد کنیم. توجه کنید که برای تبدیل رشته به `int` از متد `parseInt()` از کلاس `Integer` و برای تبدیل رشته به `double` از متد `parseDouble()` از کلاس `Double` استفاده کرده‌ایم. به طور مشابه برای تبدیل رشته به `bool` می‌توان از `Boolean.parseBoolean()` استفاده کرد.

```java
import java.io.*;

public class Main
{
    public static void main(String[] args)
    {
        try
        {
            File myFile = new File("student.txt");
            FileReader fileReader = new FileReader(myFile);
            BufferedReader reader = new BufferedReader(fileReader);
            String line;
            if ((line = reader.readLine()) != null)
            {
                String[] studentInformation = line.split(",");
                Student student = new Student(studentInformation[0],
                                              studentInformation[1],
                                              Integer.parseInt(studentInformation[2]),
                                              Integer.parseInt(studentInformation[3]),
                                              Double.parseDouble(studentInformation[4]));
                student.printInformation();
            }
            reader.close();
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

متد `split()` با دریافت یک جداکننده[7] مثل `","` رشته را به پنج رشته‌ای که بین جداکننده‌ها قرار دارند می‌شکند و خروجی را به صورت لیستی از این رشته‌ها برمی‌گرداند.

متد `printInformation()` را به صورت زیر برای کلاس `Student` تعریف می‌کنیم تا اطلاعات دانشجویی که داخل فایل بود را در خروجی به طور تمیز و منسجم چاپ کنیم.

```java
    public void printInformation()
    {
        System.out.println("First Name:  " + getFirstName());
        System.out.println("Family Name: " + getFamilyName());
        System.out.println("Age:         " + getAge());
        System.out.println("Entry Year:  " + getEntryYear());
        System.out.println("GPA:         " + getGpa());
    }
```

[7] delimiter

خروجی این برنامه به صورت زیر خواهد بود:

```
First Name:  Ryan
Family Name: Smith
Age:         19
Entry Year:  1402
GPA:         3.7
```

تمرین ۷: اطلاعات سه دانشجویی که در تمرین ۶ ایجاد کردید را از فایل `students.txt` خوانده و چاپ کنید.

نوشتن شیٔ به فایل

با نوشتن متن داخل فایل آشنا شدیم. نوشتن داده‌های متنی زمانی مناسب است که اطلاعات توسط برنامه‌ها و کاربران مختلف خوانده و بررسی شوند. مثلاً می‌توانیم داده‌های قبلی را که نوشته بودیم با یک برنامه‌ی خوانش متن باز کرده و ببینیم، یا درصورتی که در فرمت `csv` باشد آن را به وسیله‌ی یک برنامه‌ی مخصوص دیتابیس یا صفحه گسترده[8] مشاهده کنیم.

تمرین ۸: فایل CSV[9] راهی برای ذخیره‌ی داده‌های جدولی[10] است که ساختار ساده‌ای هم دارد. اولین خط آن نماینده‌ی عنوان ستون‌هاست که توسط یک جداکننده جدا می‌شوند. در CSV این جداکننده ویرگول(`,`) است، ولی استفاده از جداکننده‌های دیگر هم مجاز است؛ مثلاً در فایل‌های TSV[11] از تب استفاده می‌شود. برای نمونه محتوای یک فایل ساده‌ی `csv` که در آن اطلاعات فقط یک دانشجو (همان دانشجوی فرضی) ذخیره شده است به صورت زیر می‌باشد:

```
First Name, Family Name, Age, Entry Year, GPA
Ryan, Smith, 19, 1402, 3.7
```

برنامه‌ای بنویسید که اطلاعات دانشجوها را به فرمت `csv` در یک فایل با نام `students.csv` ذخیره کند. آنگاه اطلاعات سه دانشجویی که در تمرین ۶ ایجاد کردید را در این فایل ذخیره کرده و با یک برنامه‌ی مخصوص آن را مشاهده کنید.

راه دیگری که می‌توان برای ذخیره‌ی وضعیت اشیاء استفاده کرد سریال‌سازی[12] است. محتوای فایلی که سریال‌سازی می‌شود برای انسان قابل درک نیست، ولی برنامه راحت‌تر از فایل متنی قادر به خواندن آن خواهد بود (چرا؟). همچنین سریال‌سازی امن‌تر است، چراکه داده‌های متنی به راحتی قابل تغییر هستند، اما داده‌های سریال‌سازی شده خیر.

---

[8] spreadsheet
[9] Comma-separated values
[10] tabular data
[11] Tab-separated values
[12] serialization

برای اینکه یک شئ سریال‌سازی شده را داخل فایل بنویسیم ابتدا یک FileOutputStream می‌سازیم. آرگومان کانستراکتور این کلاس رشته‌ی نام فایل است که در صورتی که فایلی با آن نام وجود نداشته باشد، ایجاد می‌شود. FileOutputStream می‌داند که چگونه به یک فایل اتصال برقرار کرده و یا آن را بسازد.

```
FileOutputStream fileStream = new FileOutputStream("student.ser");
```

در ادامه یک ObjectOutputStream می‌سازیم که قابلیت نوشتن اشیاء را دارد ولی نمی‌تواند مستقیماً به یک فایل متصل شود. پس fileStream را به آن می‌دهیم که اصطلاحاً به این کار زنجیر کردن یک جریان به جریان دیگری[13] هم گفته می‌شود. با مفهوم جریان‌ها در ادامه بیشتر آشنا می‌شوید.

```
ObjectOutputStream os = new ObjectOutputStream(fileStream);
```

در ادامه شئ student از کلاس Student را داخل فایل می‌نویسیم. البته توجه داشته باشید درصورتی می‌توانیم این کار را بکنیم که کلاس Student، Serializable را که در پکیج java.io است پیاده‌سازی[14] کرده باشد.

```
os.writeObject(student);
```

در نهایت هم جریان را می‌بندیم، و می‌دانیم که بستن جریان مافوق، جریان‌های زیردست (مثلاً در اینجا FileOutputStream) را هم می‌بندد.

```
import java.io.*;

public class Main
{
    public static void main(String[] args)
    {
        try
        {
            Student student = new Student("Ryan", "Smith", 19, 1402, 3.7);

            FileOutputStream fileStream = new FileOutputStream("student.ser");
            ObjectOutputStream os = new ObjectOutputStream(fileStream);
            os.writeObject(student);

            os.close();
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

تمرین ۹: سه شئ student را که در تمرین ۶ ایجاد کرده بودید داخل فایل students.ser بنویسید.

[13] chaining one stream to another
[14] implement

جریان‌ها

تا به اینجای کار با جریان‌های مختلفی کار کردیم. دو نوع جریان مهم داریم: ۱) جریان اتصالی[15] و ۲) جریان زنجیری[16].

جریان‌های اتصالی مستقیماً به مقصدی برای نوشتن یا منبعی برای خواندن متصل می‌شوند که این منابع و مقاصد می‌توانند فایل‌ها، سوکت‌های شبکه یا موارد دیگر باشند. از طرفی جریان‌های زنجیری تنها زمانی کار می‌کنند که به جریان‌های دیگر وصل شوند. به طور معمول نیاز است که حداقل دو جریان به هم وصل شوند تا بتوانیم کار مفیدی انجام دهیم؛ مثل `BufferedWriter` که به `FileWriter` زنجیر شد یا `ObjectOutputStream` که به `FileOutputStream` زنجیر شد. در حالت اول می‌خواستیم عمل نوشتن را با بافرها بهینه‌تر کنیم و در حالت دوم علت زنجیر کردن این است که جریان اتصالی `FileOutputStream` سطح پایین است و متدهایی برای نوشتن بایت دارد، حال آنکه ما نمی‌خواهیم مستقیماً با بایت کار کنیم و آنها را بنویسیم؛ بلکه می‌خواهیم با اشیاء کار کرده و آنها را بنویسیم و به همین علت به جریان‌های سطح بالای زنجیری نیاز داریم.

وضعیت اشیاء

هر کدام از اشیاء وضعیتی دارند که این وضعیت همان مقادیر متغیرهای نمونه[17] هستند. با مفهوم متغیرهای نمونه در قسمت مربوط به کلاس بیشتر آشنا می‌شوید. در اینجا برای اینکه این مفهوم را بهتر متوجه شوید، به این فکر کنید که چه چیزی از کلاس `Student` هر شئ از آن را از دیگری متمایز می‌کند؟

اطلاعات دانشجوی فرضی که قبلتر تعریف کردیم را به یاد آورید:

نام: `Ryan`

نام خانوادگی: `Smith`

سن: `19`

سال ورود: `1402`

`GPA`: `3.7`

در واقع داشتن همین اطلاعات و پیاده‌سازی خود کلاس `Student` کافی است تا این شئ را دوباره بسازیم. پس این اطلاعات، یعنی نام، نام خانوادگی، سن، سال ورود و `GPA` وضعیت هر شئ از کلاس `Student` محسوب شده و هردانشجو را از دیگری متمایز

---

[15] connection stream
[16] chain stream
[17] instance variables

می‌کند. در نتیجه مقادیر متغیرهای نمونه به همراه اطلاعات دیگری که JVM برای بازسازی شئ نیاز دارد داخل فایل مقصد نوشته می‌شود.

اما اگر یک شئ ارجاعاتی به اشیاء دیگر داشته باشد چطور؟ مثلاً یکی از متغیرهای نمونه آرایه باشد، یکی دیگر شیئی از یک کلاس دیگر مثل String یا کلاسی که خودمان تعریف کرده‌ایم باشد چطور؟

پاسخ اینجاست: **وقتی شیئی سریال‌سازی می‌شود، همه‌ی اشیائی که به آن ارجاع می‌دهد هم سریال‌سازی می‌شوند. و همه‌ی اشیائی که آن اشیاء به آنها ارجاع می‌دهند هم سریال‌سازی می‌شوند. و همه‌ی اشیائی که آن اشیاء به آنها ارجاع می‌دهند هم سریال‌سازی می‌شوند. و...**

پس باید مطمئن شوید که اشیائی که ارجاع داده می‌شوند هم سریال‌سازی شدنی باشند؛ یعنی Serializable را پیاده‌سازی کنند. اگر سریال‌سازی شدنی نبودند چطور؟ می‌توانید با کلیدواژه‌ی transient مانع ذخیره‌ی هر اطلاعاتی که نمی‌خواهید مثل همین اشیائی که سریال‌سازی شدنی نیستند شوید. مثلاً در کلاس Student با تغییر زیر روی تعریف متغیر age می‌توان آن را transient کرد:

```
    transient private int age;
```

حواستان باشد؛ اشیائی که ذخیره نمی‌شوند در هنگام بازسازی شئ از روی فایل به صورت null بازگردانده می‌شوند (داده‌های اولیه[18] با مقادیر پیش‌فرضشان بازگردانده می‌شوند)، پس ممکن است اطلاعات مهمی در این بین از دست برود. البته باید به موارد دیگری هم توجه شود که در پایان به نکات مربوط به تغییر کلاس‌ها اشاره خواهد شد.

خواندن شئ از فایل

حال که توانستیم یک شئ را روی یک فایل بنویسیم، باید بتوانیم آن را از روی فایل بخوانیم. ممکن است برای خواندن در یک اجرای دیگر برنامه و یا حتی روی یک JVM دیگر بوده، ولی در عین حال انتظار داریم قابلیت بازسازی اشیاء سریال‌سازی شده که به آن دیسریالیزیشن[19] هم گفته می‌شود را داشته باشیم.

برای این کار یک FileInputStream ایجاد می‌کنیم.

```
FileInputStream fileStream = new FileInputStream("student.ser");
```

در ادامه یک ObjectInputStream ایجاد می‌کنیم که به جریان قبلی زنجیر می‌شود.

```
ObjectInputStream os = new ObjectInputStream(fileStream);
```

[18] primitive data
[19] deserialization

حال شئ داخل student.ser را خوانده و آن را در متغیر student ذخیره می‌کنیم. توجه داشته باشید که خروجی readObject() از نوع Object است و باید آن را به کلاس خودمان (در اینجا Student) کست[20] کنیم.

```
Student student = (Student) os.readObject();
```

باز هم در پایان جریان os را می‌بندیم.

```
import java.io.*;

public class Main
{
    public static void main(String[] args)
    {
        try
        {
            FileInputStream fileStream = new FileInputStream("student.ser");
            ObjectInputStream os = new ObjectInputStream(fileStream);
            Student student = (Student) os.readObject();

            student.printInformation();

            os.close();
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
```

تمرین ۱۰: سه شئ داخل students.ser را که در تمرین ۹ نوشتید خوانده و اطلاعات آنها را نمایش دهید.

نکته: اگر JVM نتواند کلاس Student را پیدا کند و یا آن را بارگذاری کند، خطای I/O می‌دهد و اگر در زمان کست کردن کلاس Student پیدا نشود، خطای پیدا نشدن کلاس داده می‌شود؛ پس امکان دارد با دو نوع خطای مختلف روبه‌رو شویم و بنابراین در قسمت catch از والد همه‌ی خطاها یعنی Exception به جای یک خطای خاص‌تر مثل IOException استفاده می‌کنیم.

نکته: متغیرهای static کلاس به ازای کلاس تعریف می‌شوند و نه به ازای هر شئ؛ در نتیجه سریال‌سازی نمی‌شوند. (با این متغیرها در ادامه‌ی درس بیشتر آشنا خواهید شد.)

برخی از تغییراتی روی کلاس که باعث صدمه دیدن در فرآیند دیسریالیزیشن می‌شود:

- حذف یک متغیر نمونه
- تغییر نوع مشخص شده‌ی یک متغیر نمونه

[20] cast

- `transient` کردن متغیر نمونه‌ای که در ابتدا چنین نبوده
- تغییر یک کلاس به گونه‌ای که دیگر سریال‌سازی شدنی نباشد
- `static` کردن یک متغیر نمونه

برخی از تغییراتی روی کلاس که باعث صدمه دیدن در فرآیند دیسریالیزیشن نمی‌شود:

- اضافه کردن متغیرهای نمونه‌ی جدید به کلاس (مقادیر پیش‌فرض در هنگام دیسریالیزیشن برای متغیرهای نمونه‌ای که در هنگام فرآیند سریال‌سازی وجود نداشتند ایجاد می‌شود)
- حذف `transient` برای متغیر نمونه‌ای که در ابتدا `transient` بوده (اشیاء سریال‌سازی شده‌ی قبلی مقدار پیش فرض برای چنین متغیرهایی می‌گیرند)

در صورتی که کلاس شما در طول زمان دچار تغییر می‌شود باید شناسه‌ی `serialVersionUID` را هم که در دیسریالیزیشن توسط `JVM` استفاده می‌شود در کلاس قرار دهید. این شناسه با دستور خط فرمان زیر برای کلاس به دست می‌آید:

```
$ serialver -classpath <classpath> Student
```

به جای `<classpath>` آدرس دایرکتوری که `Student.class` در آن قرار دارد را جایگذاری کنید.

در اینجا ما متغیر `age` را `transient` کرده و شناسه‌ی کلاس را به صورت زیر به دست آورده‌ایم:

```
➜ serialver -classpath out/production/java-sample-code/ Student
Student:    private static final long serialVersionUID = -7020568467139577O7L;
```

اگر شئ را بنویسیم و بخوانیم مشکلی نخواهیم داشت. اما اگر شئ را بنویسیم و قبل از خواندن آن، `age` را دوباره به حالت قبل برگردانیم، به مشکل می‌خوریم. برای اینکه به مشکل نخوریم باید متغیر `serialVersionUID` را هم به کلاس اضافه کنیم:

```java
import java.io.Serializable;

public class Student implements Serializable
{
    private static final long serialVersionUID = -7020568467139577O7L;
    private String firstName;
    private String familyName;
    private int age;
    private int entryYear; // four-digit number
    private double gpa;    // on scale of 4
```

```
    public Student(String firstName, String familyName, int age, int entryYear, double gpa)
    {
        this.firstName = firstName;
        this.familyName = familyName;
        this.age = age;
        this.entryYear = entryYear;
        this.gpa = gpa;
    }

    public void printInformation()
    {
        System.out.println("First Name:  " + getFirstName());
        System.out.println("Family Name: " + getFamilyName());
        System.out.println("Age:         " + getAge());
        System.out.println("Entry Year:  " + getEntryYear());
        System.out.println("GPA:         " + getGpa());
    }

    public String getFirstName()
    {
        return firstName;
    }

    public String getFamilyName()
    {
        return familyName;
    }

    public int getAge()
    {
        return age;
    }

    public int getEntryYear()
    {
        return entryYear;
    }

    public double getGpa()
    {
        return gpa;
    }
}
```