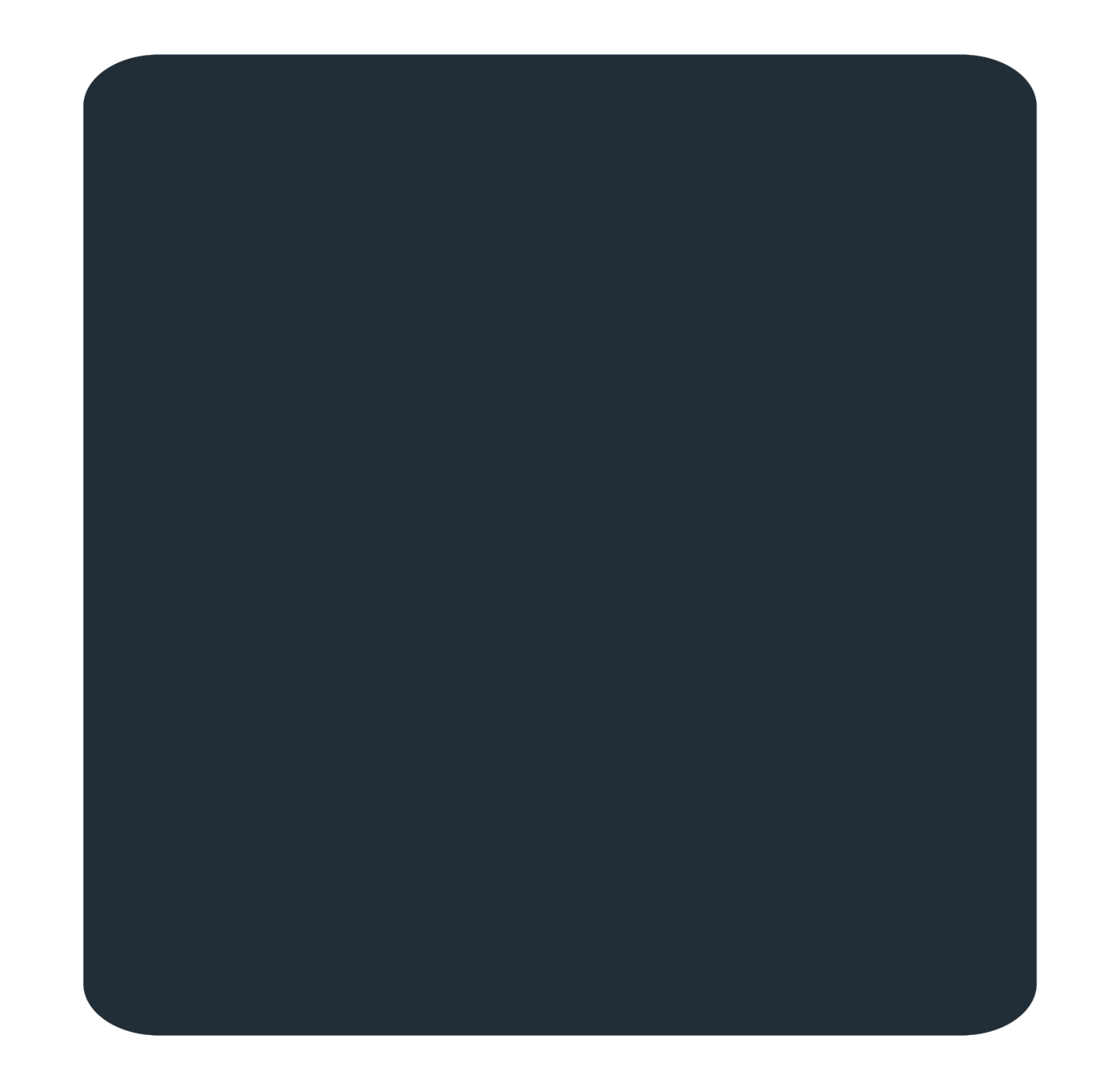


کارگاه برنامه نویسی پیشرفته

دستور کار شماره هشت

اهداف

* آشنایی با اکسپشن‌ها
* آشنایی با پرتاب کردن اکسپشن‌ها
* مدیریت خطاها و اکسپشن‌ها با try-catch
* انواع اکسپشن‌ها
* ساختن اکسپشن اختصاصی خودمان
* آشنایی با IO استریم‌ها
* معرفی FileStream و ObjectStream
* آشنایی با try with resources



فهرست مطالب

[اکسپشن‌ها ۳](#_اکسپشن‌ها)

[try-catch](#_Try-catch)  4

[اکسپشن‌های چک شده و چک نشده](#_اکسپشن_های_چک) 5

[throws ۶](#_Throws)

[Stream](#_Stream) 8

[File Reader & File Writer](#_FileReader_&_FileWriter) ۹

[Object Serialization](#_Object_Serialization) 11

[Serializable](#_Serializable) 11

[ObjectOutputStream 12](#_exywrfk05x72)

[ObjectInputStream 13](#_e0bq459hz0tk)

[Transient](#_Transient) ۱۴

[Try With Resources 15](#_Try_with_Resources_1)

[دفترچه یادداشت 18](#_دفترچه_یادداشت)

# **اکسپشن‌ها[[1]](#footnote-1)**

منظور از اکسپشن‌ها، رخدادهای غیرمنتظره و ناخواسته‌ای است که جریان و روند اجرای طبیعی برنامه را مختل می‌کنند و باعث توقف اجرای برنامه می‌شوند. برای مثال، فرض کنید یک برنامه برای ثبت‌نام تعدادی دانش‌آموز در سامانه‌ای نوشته‌اید، که این برنامه شامل متدی برای دریافت کد ملی دانش‌آموزان است:

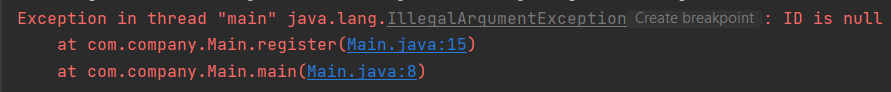
|  |
| --- |
| public static void register(String id) |

در شرایطی که ورودی این متد null باشد برنامه با مشکل مواجه می‌شود:

|  |
| --- |
| public static void register(String id) {  if(id==null)  {  throw new IllegalArgumentException("ID is null");  }  else  {  //....  } } |

در چنین شرایطی، متد یک شی از نوع اکسپشن IllegalArgument و با پیغام مورد نظر پرتاب[[2]](#footnote-2) کرده و اجرای برنامه را متوقف می‌کند.

|  |
| --- |
| public static void main(String[] args) {  register(null);  System.out.println("Test"); } |

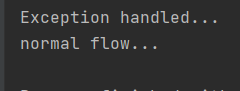


همانطور که مشاهده می‌کنید "Test" چاپ نشد و برنامه با دریافت اکسپشن در متد register، متوقف شد.  
برای جلوگیری از متوقف شدن برنامه هنگام رخ دادن اکسپشن‌ها نیاز به هندل[[3]](#footnote-3) کردن آن‌ها داریم و برای این کار از try-catch استفاده می‌کنیم.

# **Try-catch**

در بلوک try قسمتی از کد را قرار می‌دهیم که اکسپشن می‌تواند رخ دهد. در بلوک catch نیز عملیات مورد نظرمان برای مدیریت کردن این اکسپشن را قرار می‌دهیم. همچنین نوع اکسپشن‌های مورد نیاز برای هندل کردن باید به عنوان ورودی به catch داده شود.

|  |
| --- |
| public static void main(String[] args) {  try {  register(null);  } catch (IllegalArgumentException e) {  System.out.println("Exception handled...");  }  System.out.println("normal flow...");  } |

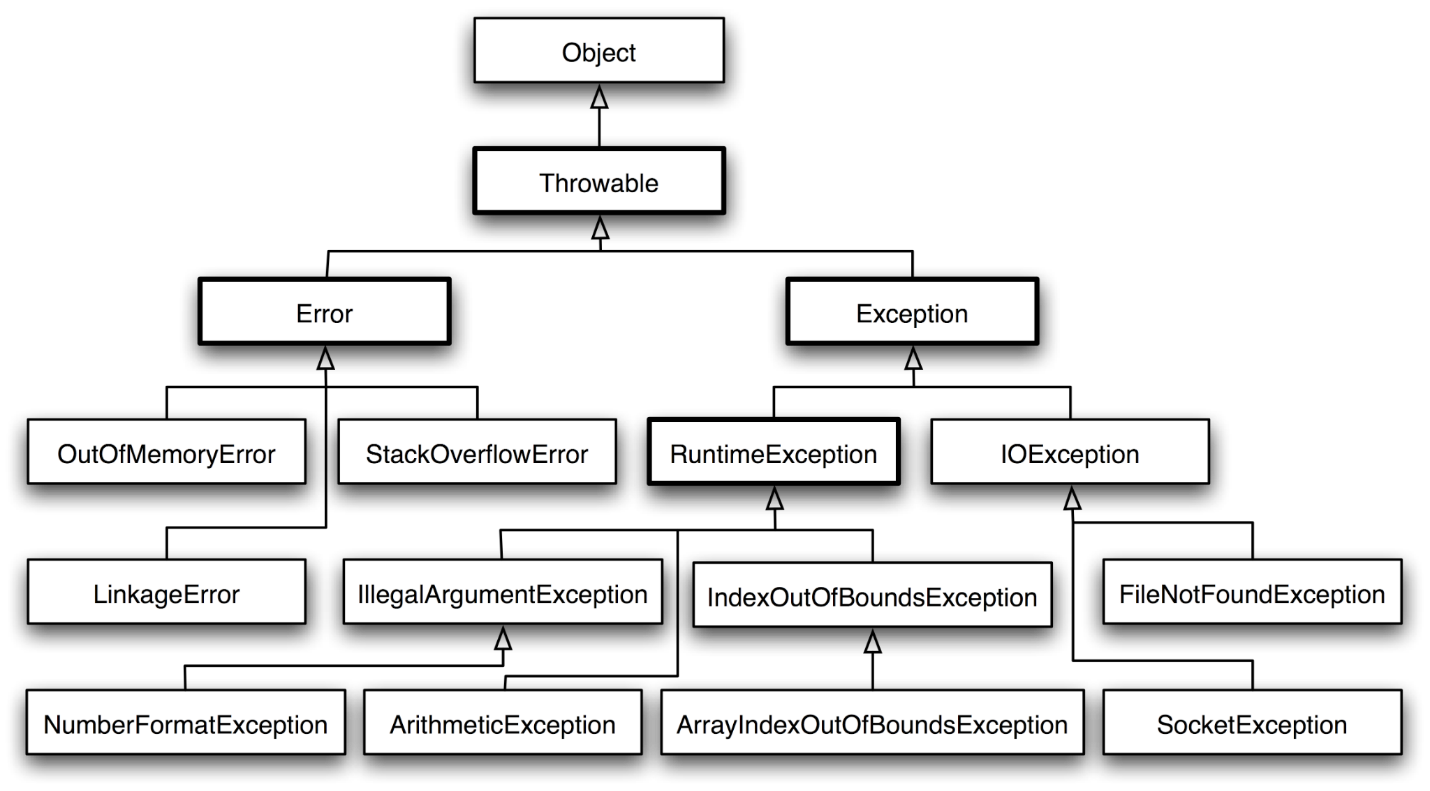


همانطور که مشاهده میکنید، این‌بار اکسپشن تولید شده، باعث توقف برنامه نشد.

برای catch می‌توان از چندین اکسپشن هم استفاده کرد[[4]](#footnote-4):

|  |
| --- |
| *catch (IllegalArgumentException | ArithmeticException e)* |

## اکسپشن‌های چک شده[[5]](#footnote-5) و چک نشده



* اکسپشن‌هایی که از کلاس RuntimeException ارث‌بری می‌کنند، از نوع چِک نشده محسوب می‌شوند و مربوط به مشکلات در منطق برنامه هستند. در نتیجه اکسپشن‌هایی غیر پیش بینی شده به شمار می‌روند.
* اکسپشن‌هایی که از کلاس RuntimeException ارث‌بری نمی‌کنند، چِک شده به شمار می‌روند و مدیریت نکردن این اکسپشن‌ها، موجب Compile Error می‌شود. این نوع از اکسپشن‌ها قابل پیشبینی هستند، مانند:FileNotFoundException

با استفاده از همین ارث‌بری‌ها می‌توانیم اکسپشن‌های مورد نیاز خودمان را هم بسازیم. فرض کنید برنامه‌ای برای ثبت نمرات دانش آموزان ساخته‌اید و می‌خواهید در صورتی که دانش آموزی در لیست یافت نشد یک اکسپشن پرتاب بشود (از جنس چِک شده):

|  |
| --- |
| class StudentNotFoundException extends Exception {  public StudentNotFoundException(String message) {  super(message); // When Exception occured }  }  } |

برای مطالعه بیشتر درباره ساختن اکسپشن‌ها می‌توانید به لینک مقابل مراجعه نمایید:

<https://stackify.com/java-custom-exceptions>

**برای مدیریت کردن اکسپشن‌ها دو راه حل وجود دارد**:

1- استفاده از try-catch

2- استفاده از throws در signature متد مورد نظر

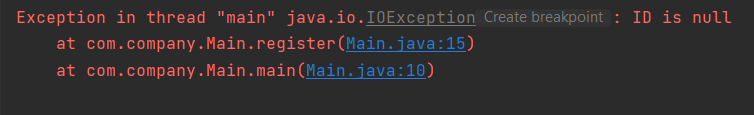
# **Throws**

با استفاده از throws مشخص می‌کنیم که در متد چه اکسپشن‌هایی پرتاب شود و در حقیقت وظیفه مدیریت کردن این اکسپشن‌ها (با try-catch)‌ به فراخواننده این متد سپرده می‌شود (که می‌تواند یک متد دیگر و یا خود JVM باشد؛ که در حالت دوم، اگر اکسپشن رخ بدهد برنامه متوقف می‌شود و بازیابی وجود ندارد).

نکته: کلمه throws برای متدهایی که اکسپشن از نوع چِک شده پرتاب می‌کنند حتما باید استفاده شود. استفاده از throws برای اکسپشن‌های چِک نشده الزامی ندارد.

|  |
| --- |
| public static void main(String[] args) throws IOException {  register(null); } public static void register(String id) throws IOException {  if(id==null)  {  throw new IOException("ID is null");  }  else  {  //...  }  } |

در این حالت چون متد دیگری وجود ندارد که متد main را صدا بزند، IOException پرتاب می‌شود و چون با try-catch هندل نشده است، منجر به توقف برنامه می‌شود.



|  |
| --- |
| public static void main(String[] args) {  try {  register(null);  } catch (IOException exception) {  System.err.println("null id in register method");  } } public static void register(String id) throws IOException {  if(id==null)  {  throw new IOException("ID is null");  }  else  {  //...  }  } |



*(حالت هندل شده)*

Java I/O

همان‌طور که می‌دانید، تمام متغیرها و اشیاء که در طول برنامه از آن‌ها استفاده می‌کنیم، در حافظه رم[[6]](#footnote-6) ذخیره می‌شوند و همچنین می‌دانیم که رم یک حافظه پرسرعت اما فرار[[7]](#footnote-7) است. در نتیجه هنگامی که کامپیوتر را خاموش کنیم، تمام داده‌‌های ذخیره شده روی آن حذف خواهند شد و امکان دسترسی مجدد به آن‌ها را نخواهیم داشت. اما خیلی وقت‌ها نیاز است که مطالب به صورت بلند مدت ذخیره شوند. در این مواقع باید داده‌های خود را به صورت فایل و بر روی یک حافظه ثانویه[[8]](#footnote-8) مانند HDD ذخیره کنیم. در جاوا این کار توسط java io صورت می‌پذیرد. همچنین تمام کلاس‌های مورد نیاز برای این کار در پکیج java.io موجود هستند.

# **Stream**

در جاوا استفاده از I/O توسط مفهومی به نام استریم[[9]](#footnote-9) صورت می‌گیرد. یک استریم، جریانی از داده‌ها است که بین مبدا و مقصد ساخته شده و باعث انتقال داده‌ها می‌شود.

هنگام اجرا شدن یک برنامه جاوا، سه استریم به صورت خودکار ایجاد می‌شوند که با آن‌ها آشنایی دارید:

* System.in
* System.out
* System.err

به‌طور کلی دو نوع استریم وجود دارد: استریم‌های مبنی بر بایت[[10]](#footnote-10) که ورودی و خروجی را به صورت باینری انجام می‌دهد و استریم‌های مبنی بر کاراکتر[[11]](#footnote-11) که با توالی از کاراکترها کار می‌کند.

در این دستورکار قصد داریم FileReader و FileWriter را به عنوان استریم‌های کاراکتری، و ObjectInputStream و ObjectOutputStream را به عنوان استریم‌های باینری شرح دهیم.

## FileReader و FileWriter

توجه کنید که در جاوا دو نوع استریم ورودی و خروجی داریم و نمی‌توانیم تنها با یک استریم هر دو عمل را انجام دهیم. FileWriter نقش خروجی و FileReader نقش ورودی را خواهد داشت.

هنگامی که می‌خواهیم کار با یک فایل را شروع کنیم کافیست یک استریم از نوع مورد نظر بسازیم و در کانستراکتور[[12]](#footnote-12) آن، فایل مورد نظر را با استفاده از آدرس یا یک شئ از نوع File صدا بزنیم. در این حالت استریم مورد نظر ساخته خواهد شد و فایل ما آماده نوشتن یا خواندن خواهد بود.

فرض کنید می‌خواهیم یک فایل متنی با فرمت .txt ایجاد کنیم و داخل آن متنی کوتاه را بنویسیم. با دقت کد زیر را مطالعه کنید و در ادامه به نکات گفته شده توجه کنید:

|  |
| --- |
| import java.io.FileWriter; import java.io.IOException;  public class Main {  public static void main(String[] args) {   // You must handle the exception of opening a file.  try {  FileWriter fileWriter = new FileWriter("test.txt");   // You can write a string, char or array of chars with FileWriter  fileWriter.write("This is a test string. \nThis is a new line.");   // You must close your streams at the end  fileWriter.close();   } catch (IOException e) {  e.printStackTrace();  }  } } |

1. بعد از استفاده از یک استریم باید حتما آن را با متد close ببندید. در غیر این صورت تضمینی برای ذخیره شدن داده‌ها به طور صحیح وجود نخواهد داشت.
2. باز کردن یک فایل، اکسپشن از نوع چِک شده ایجاد می‌کند. به همین دلیل باید مدیریت اکسپشن مناسب انجام شود.
3. اگر فایلی را به قصد نوشتن باز کنیم و این فایل از قبل وجود داشته باشد، تمام داده‌های آن از بین خواهد رفت[[13]](#footnote-13). به عنوان مثال فرض کنید در کد فوق، از قبل فایلی به نام test.txt در دایرکتوری موجود باشد که شامل یک متن است. پس از اجرای برنامه بالا، تمام متن قبلی آن فایل پاک شده و تنها متن مشخص شده در برنامه باقی خواهد ماند. در ادامه، راه‌حلی برای این مشکل ارائه شده است.
4. استریم FileWriter دارای یک متد با نام write است که وظیفه آن نوشتن داده‌ی کاراکتری است و می‌توانیم به این متد داده‌هایی از نوع رشته[[14]](#footnote-14)، کاراکتر[[15]](#footnote-15) و آرایه‌ای از کاراکتر[[16]](#footnote-16) پاس دهیم.

حال فرض کنید یک فایل با نام test.txt در دایرکتوری موجود است و متنی هم در آن نوشته شده است. قصد داریم با استفاده از FileReader محتوای این فایل را بخوانیم. به کد زیر توجه کنید:

|  |
| --- |
| import java.io.IOException; import java.io.FileReader;  public class Main {  public static void main(String[] args) {    try {  FileReader fileReader = new FileReader("test.txt");   int i;  while((i = fileReader.read()) != -1) {  System.out.print((char)i);  }   fileReader.close();   } catch (IOException e) {  e.printStackTrace();  }  } } |

متد read یک کاراکتر از فایل می‌خواند، اما باید توجه کنید که نوع آن به صورت int است. اگر مقدار int برابر «1-»، باشد به انتهای فایل رسیده‌ایم و کار تمام می‌شود؛ در غیر این صورت، برای نمایش کاراکتر خوانده شده کافیست آن را به char تبدیل کنیم.

## Object Serialization

در جاوا این امکان وجود دارد که یک شئ را به صورت کامل در یک فایل بنویسیم یا از یک فایل بخوانیم. برای این کار ابتدا شئ مورد نظر باید به صورت توالی از بایت‌ها نوشته شود که شامل اطلاعات شئ مورد نظر و همچنین اطلاعات کلاس آن می‌شود. به این فرآیند در جاوا object serialization گفته می‌شود. هنگام نوشتن شئ در فایل، آن شئ باید serialized شود و هنگام خواندن آن، باید شئ deserialized شود.

### **Serializable**

یک اینترفیس است که هیچ متدی برای پیاده‌ سازی ندارد و صرفا برای نشانه گذاری کلاس‌هایی به کار می‌رود که قرار است در فایل ذخیره شوند. کلاسی که قصد داریم اشیاء آن را در فایل ذخیره کنیم باید این اینترفیس را پیاده‌سازی کند در غیر این صورت امکان ذخیره اشياء وجود نخواهد داشت. توجه کنید که علاوه بر خود شئ، تمام فیلدهای آن هم باید serializable باشند.

نکته: تمام داده‌های نوع اولیه به صورت پیشفرض serializable هستند. بقیه کلاس‌ها باید به صورت جداگانه بررسی شوند که برای این کار می‌توانید به مستندات جاوا مراجعه کنید.

در کد زیر قصد داریم یک کلاس با نام Person بسازیم که شامل اطلاعات یک فرد می‌شود و آن را serializable کنیم.

|  |
| --- |
| import java.io.Serializable;  public class Person implements Serializable {    private String name;  private String id;  private int age;   public Person(String name, String id, int age) {  this.name = name;  this.id = id;  this.age = age;  }   @Override  public String toString() {  return "Name: " + name + "\nID: " + id + "\nAge: " + age;   } } |

### **ObjectOutputStream**

این کلاس برای نوشتن اشیاء serializable داخل فایل استفاده می‌شود. کانستراکتور آن، یک استریم را، که زیرکلاس OutputStream است، دریافت می‌کند. در این مثال از FileOutputStream استفاده می‌کنیم.

کلاس FileOutputStream فایلی که قرار است اشیاء را داخل آن بنویسیم، مشخص می‌کند.

|  |
| --- |
| import java.io.FileOutputStream; import java.io.IOException; import java.io.ObjectOutputStream;  public class Output {  public static void main(String[] args) {   Person p = new Person("Ali", "123456789", 24);   try {  FileOutputStream fOut = new FileOutputStream("test.bin");  ObjectOutputStream out = new ObjectOutputStream(fOut);   out.writeObject(p);   fOut.close();  out.close();   } catch (IOException e) {  e.printStackTrace();  }  } } |

در این مثال پسوند bin یک پسوند اختیاری است که معمولا برای فایل‌های باینری مورد استفاده قرار می‌گیرد.

متد writeObject کار نوشتن شئ p را در فایل مورد نظر انجام می‌دهد. پس از اجرای برنامه بالا یک فایل با نام test.bin ایجاد خواهد شد که شامل اطلاعات شئ p است.

### **ObjectInputStream**

این کلاس برای deserialize کردن و خواندن اشیاء استفاده می‌شود. کانستراکتور این کلاس، یک استریم را که زیرکلاس InputStream است، دریافت می‌کند. ما در این مثال از FileInputStream استفاده می‌کنیم.

در کد زیر قصد داریم اطلاعات p را که در کد قبل داخل test.bin نوشتیم deserialize کرده و اطلاعات آن را نمایش دهیم.

|  |
| --- |
| import java.io.FileInputStream; import java.io.IOException; import java.io.ObjectInputStream;  public class Input {  public static void main(String[] args) {   try {  FileInputStream fIn = new FileInputStream("test.bin");  ObjectInputStream in = new ObjectInputStream(fIn);   Person p = (Person)in.readObject();   System.out.println(p);   fIn.close();  in.close();   } catch (IOException e) {  e.printStackTrace();  } catch (ClassNotFoundException e) {  e.printStackTrace();  }  } } |

توجه کنید که خروجی متد readObject یک شئ از نوع Object است. به همین خاطر باید قبل از استفاده، آن را به نوع مورد نظر (در اینجا Person) تبدیل کنیم.

توجه: در یک برنامه ObjectOutputStream حتما باید قبل از ObjectInputStream ساخته شود در غیر این صورت با مشکل مواجه خواهید شد. علت این موضوع در مستندات جاوا شرح داده شده است.

### **Transient**

گاهی اوقات پیش می‌آید که قصد نداریم بخشی از اطلاعات یک شئ را ذخیره کنیم یا به خاطر serializable نبودن بخشی از شئ امکان این کار وجود ندارد. در این موقعیت با استفاده از کلمه کلیدی transient پیش از تعریف فیلد مورد نظر، می‌توانیم باعث serialize نشدن آن شویم.

توجه کنید هنگامی که یک شئ که بخشی از آن transient باشد را از روی فایل می‌خوانیم، مقدار فیلدهای transient شده برابر مقدار پیش فرض خواهد بود.

به نکات مهم زیر در مورد کار با فایل توجه کنید:

1. شما تنها می‌توانید یک استریم ورودی و یک استریم خروجی به یک فایل داشته باشید. ساختن استریم‌های بیش‌تر باعث ایجاد مشکلاتی در کار با فایل‌ها خواهد شد.
2. امکان استفاده از اشیاء serialize شده توسط برنامه‌های مختلف در پلتفرم‌های متفاوت وجود دارد؛ اما باید این را در نظر بگیرید که کلاسی که می‌خواهید از اشیاء آن استفاده کنید باید در تمام پلتفرم‌ها یکسان باشد. به عنوان مثال اگر کلاس Person که در مثال‌های بالا از آن استفاده کردیم در چند کامپیوتر مختلف وجود داشته باشد، امکان استفاده از فایل ذخیره شده در تمام کامپیوترها وجود دارد.

# **Try with Resources**

به قطعه کد زیر توجه کنید:

|  |
| --- |
| try {  FileOutputStream out = new FileOutputStream(filename);  String text = "hello world!";  out.write(text.getBytes());  out.close(); } catch (FileNotFoundException e) {  System.out.println("Could not find the file!"); } catch (IOException e) {  System.out.println("I/O error occurred!"); } |

همانطور که می‌دانید، اگر حین ساختن FileOutputStream یا نوشتن در آن اکسپشن پرتاب شود، وارد یکی از بلاک‌های catch می‌شویم. در این حالت، مابقی بلاک try اجرا نمی‌شود و استریم ما بسته نخواهد شد!

برای رفع این مشکل، می‌توان از بلاک finally استفاده کرد:

|  |
| --- |
| FileOutputStream out = null; try {  out = new FileOutputStream(filename);  String text = "hello world!";  out.write(text.getBytes()); } catch (FileNotFoundException e) {  System.out.println("Could not find the file!"); } catch (IOException e) {  System.out.println("I/O error occurred!"); } finally {  if (out != null) {  try {  out.close();  } catch (IOException e) {  System.out.println("I/O error occurred while closing the stream!");  }  } } |

اما بلاک finally چیست؟ این بلاک تحت هر شرایطی از جمله:

1. اجرا شدن کل بلاک try و رخ ندادن اکسپشن

۲. catch شدن یک اکسپشن

۳. وجود دستور return در هر یک از بلاک‌های try-catch

اجرا خواهد شد.

همانطور که مشاهده می‌کنید، در این حالت try-catch دیگری به کد اضافه شده و آن را طولانی‌تر و دیباگ کردن آن را سخت‌تر کرده است.

نکته قابل توجه دیگر این است که هنگام بستن یک استریم که از قبل بسته شده، IllegalStateException پرتاب می‌شود.

try with resources تمام مشکلات فوق، از جمله بستن استریم‌ها را حل می‌کند. در این مکانیزم، یک یا چند منبع[[17]](#footnote-17) را مانند شکل زیر داخل دستور try اعلام می‌کنیم:

|  |
| --- |
| try (FileOutputStream out = new FileOutputStream(filename)) {  String text = "hello world!";  out.write(text.getBytes()); } catch (FileNotFoundException e) {  System.out.println("Could not find the file!"); } catch (IOException e) {  System.out.println("I/O error occurred!"); } |

حال اگر اکسپشن پرتاب و یا بلاک try به طور کامل اجرا شود، منابع به صورت خودکار بسته می‌شوند (دلیل: [AutoClosable](https://docs.oracle.com/javase/8/docs/api/java/lang/AutoCloseable.html)).

# **نکته**

در مثال‌های زیر، دقت کنید که Exceptionهای خود را به صورت جزئی catch یا پرتاب کنید.

**الف)** در تعریف متد[[18]](#footnote-18)ها و هنگام پرتاب کردن اکسپشن:

فرض کنید که هم‌تیمی شما در یک پروژه تعدادی متد تعریف کرده و شما می‌خواهید از این متدها استفاده کنید. بدیهی است که تعریف متد اول، برای شما واضح نخواهد بود. تعریف صحیح، تعریف متد دوم است که در آن نام اکسپشن مورد نظر، ذکر شده است.

|  |
| --- |
| public void doNotDoThis() throws Exception {  }  public void doThis() throws IllegalArgumentException {  } |

**ب)** برای catch کردن اکسپشن‌ها، دو روش زیر را در نظر بگیرید:

#### روش اول)

|  |
| --- |
| try (FileOutputStream out = new FileOutputStream(filename)) {  String text = "some text";  out.write(text.getBytes()); } catch (Exception e) {  e.printStackTrace(); } |

#### روش دوم)

|  |
| --- |
| try (FileOutputStream out = new FileOutputStream(filename)) {  String text = "some text";  out.write(text.getBytes()); } catch (FileNotFoundException e) {  System.out.println("Could not find the file!");  // do a specific task (eg: try to open another file, ...) } catch (IOException e) {  System.out.println("I/O error occurred!");  // do a specific task (eg: display an error message to the user, ...) } |

واضح است که در روش دوم، نه تنها خوانایی کد بالاتر می‌رود، بلکه شما می‌توانید اکسپشن‌های احتمالی برنامه خود را حالت بندی کرده و برای هر کدام، راه‌حل‌‌های متفاوتی[[19]](#footnote-19) ارائه دهید. در این روش به ترتیب catch کردن اکسپشن‌ها دقت کرده و با توجه به اصول وراثت و سلسله مراتب کلاس‌های اکسپشن، ابتدا اکسپشن‌های فرزند را catch کنید.

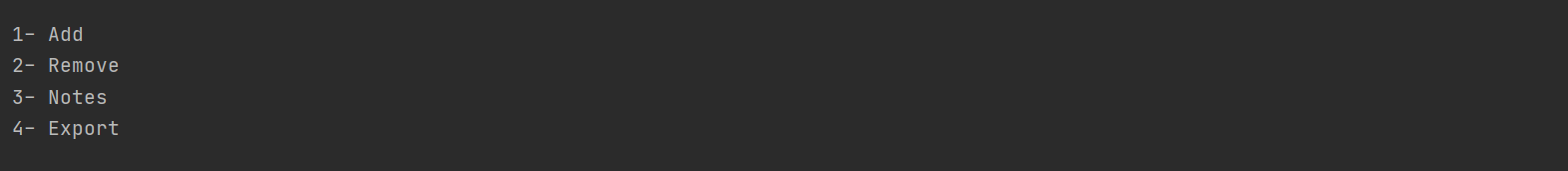
# دفترچه یادداشت

در این جلسه قصد داریم یک دفترچه یادداشت شخصی در محیط کنسول، پیاده‌سازی کنیم.

فراموش نکنید که کدهای خود را بر روی یک ریپازیتوری با نام «AP-Workshop8-Spring-2022» قرار دهید.

این دفترچه یادداشت، باید قابلیت‌های زیر را داشته باشد:

* اضافه کردن یک یادداشت جدید
* حذف کردن یک یادداشت دلخواه
* مشاهده لیستی از تمام یادداشت‌ها
* گرفتن خروجی از یک یادداشت دلخواه



در ادامه برای هر بخش، توضیحاتی خواهیم داد.

## Add

* کاربر بتواند برای هر یادداشت جدید یک اسم دلخواه انتخاب کند (اگر اسم انتخاب شده تکراری بود،

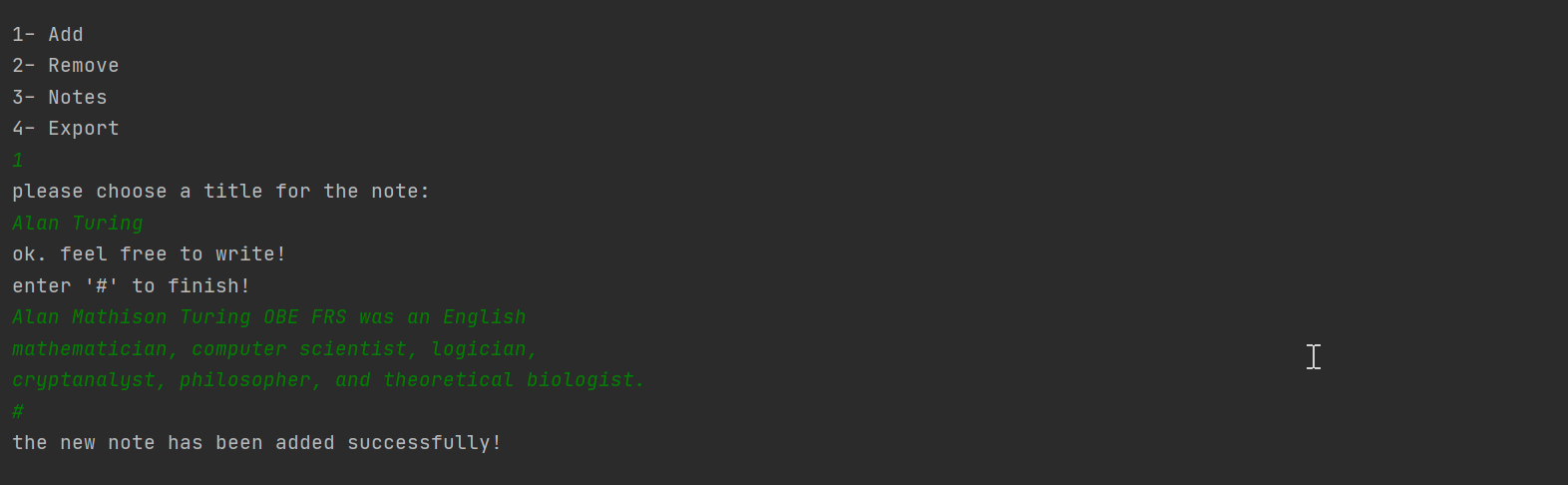
عکس العمل مناسبی رخ دهد).

* کاربر بتواند یک یادداشت جدید، که می‌تواند شامل یک یا چندین خط باشد، بنویسد.
* یادداشت جدید ایجاد شده در جایی ذخیره شود؛ به طوری که حتی در اجراهای بعدی، کاربر بتواند به

به آن دسترسی داشته باشد.

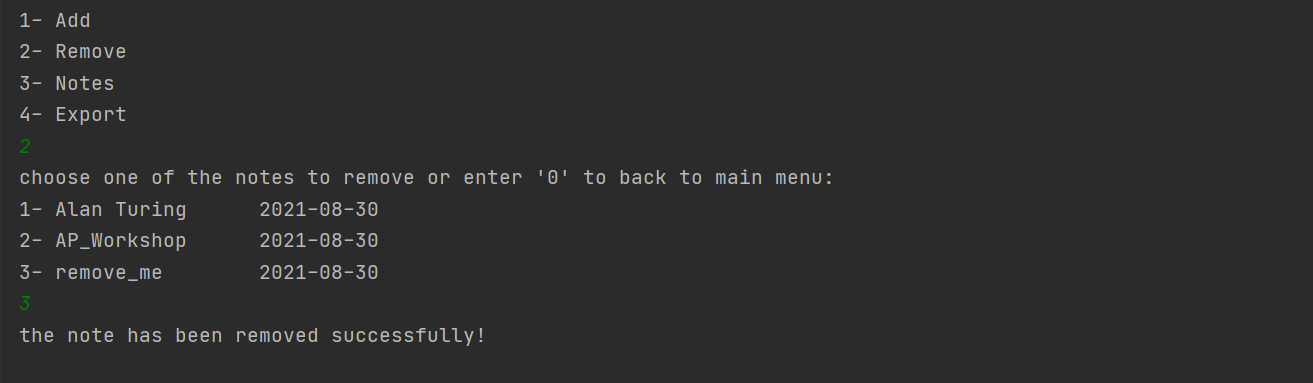
انتظار می‌رود کاربر فقط از طریق اجرای برنامه بتواند به یادداشت‌ها دسترسی داشته باشد. پس توجه کنید که یادداشت‌ها نباید در فایل txt ذخیره شوند!

* تاریخ ایجاد یادداشت در جایی به همراه یادداشت ذخیره شود (ذخیره روز، ماه و سال ایجاد یادداشت کافیست).



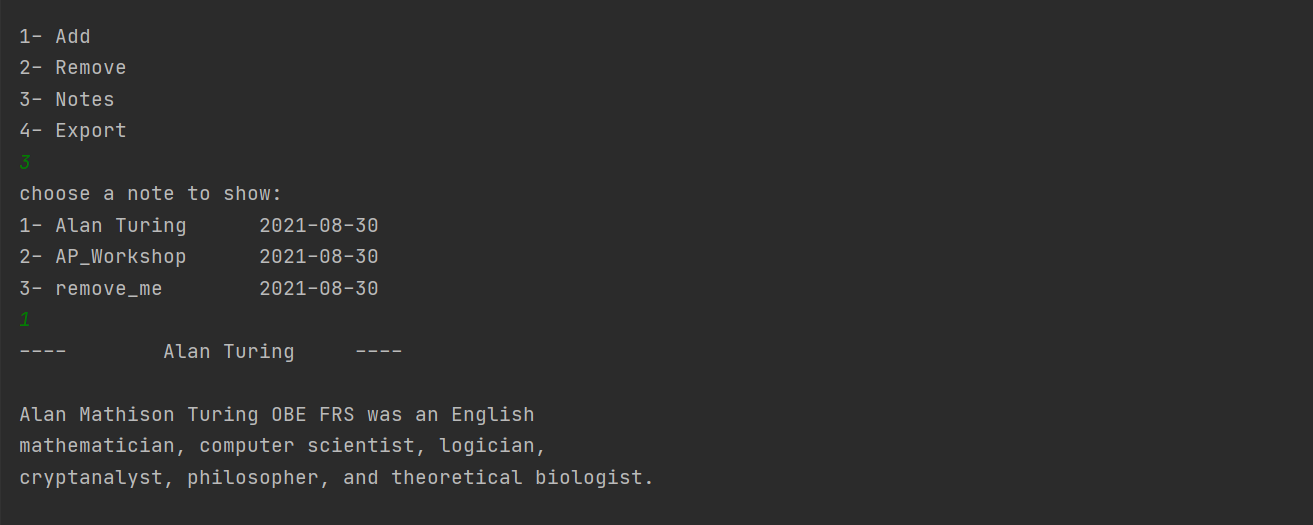
## Remove

* لیستی از یادداشت‌های ذخیره شده نمایش داده شود و کاربر با انتخاب یکی از یادداشت‌ها، بتواند آن را حذف کند.



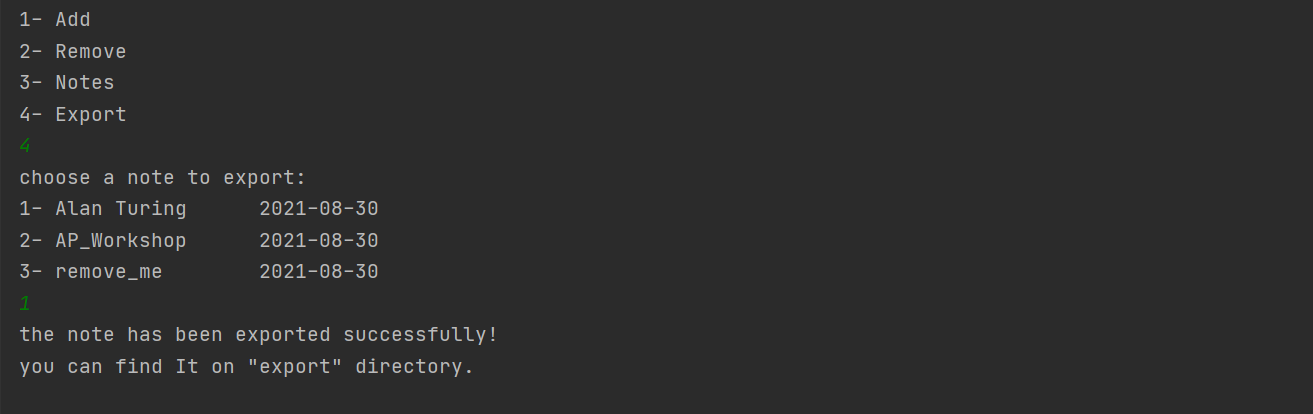
## Notes

* لیستی از تمام یادداشت‌های ایجاد شده نمایش داده شود. سپس کاربر با انتخاب یک یادداشت دلخواه، بتواند محتوای داخل آن را مشاهده کند.



## Export

* لیستی از تمام یادداشت‌های ایجاد شده نمایش داده شود و کاربر بتواند یکی از یادداشت‌ها را برای خروجی گرفتن انتخاب کند. سپس، از یادداشت انتخاب شده، یک خروجی گرفته شود و در یک فولدر مناسب (برای مثال export) ذخیره شود. خروجی باید یک فایل txt باشد. اسم فایل txt نیز همان عنوان یادداشت است.



## نکات کلی:

* با توجه به اینکه در جلسه قبل، اصول طراحی را آموختید، طراحی این برنامه نیز به عهده شما خواهد بود.
* سعی کنید از تمام آموخته‌های خود در این جلسه، برای پیاده‌سازی این تمرین استفاده کنید.

هدف این تمرین علاوه بر ایجاد تسلط نسبی شما بر روی کار با فایل، Exception handling نیز می‌باشد؛ پس انتظار می‌رود ورودی‌های اشتباه کاربر را نیز هندل کنید.

* توجه کنید که در هر یک از چهار قسمت بالا، کاربر باید بتواند به صفحه اصلی بازگردد.

برای مثال، اگر کاربر وارد قسمت Add شد، در‌صورت پشیمانی، بتواند به صفحه اصلی باز‌گردد، بدون آنکه یادداشت جدیدی اضافه کند.

* لازم به ذکر است که برنامه تمام نخواهد شد، مگر آنکه کابر اقدام به بستن برنامه کند (به این منظور، می‌توانید یک گزینه «exit» نیز به برنامه اضافه کنید).
* در‌ضمن، عکس‌هایی که مشاهده کردید صرفا یک نمونه پیاده‌سازی برای فهم بهتر سوال بود. شما مختارید که پیاده‌سازی را به هر نحو که دوست دارید انجام دهید. فقط باید شروط کلی را رعایت کنید.

1. exception [↑](#footnote-ref-1)
2. throw [↑](#footnote-ref-2)
3. handle [↑](#footnote-ref-3)
4. multi-catch [↑](#footnote-ref-4)
5. Checked Exceptions [↑](#footnote-ref-5)
6. RAM [↑](#footnote-ref-6)
7. volatile [↑](#footnote-ref-7)
8. secondary storage [↑](#footnote-ref-8)
9. stream [↑](#footnote-ref-9)
10. byte-based stream [↑](#footnote-ref-10)
11. character-based bytes [↑](#footnote-ref-11)
12. constructor [↑](#footnote-ref-12)
13. Data Truncate [↑](#footnote-ref-13)
14. string [↑](#footnote-ref-14)
15. char [↑](#footnote-ref-15)
16. char array [↑](#footnote-ref-16)
17. resource [↑](#footnote-ref-17)
18. Method Signature [↑](#footnote-ref-18)
19. recovery Actions [↑](#footnote-ref-19)