code
tutors

Java I/O

Input/Output

Working with files

Today

- You will learn to operate with streams associated with I/O devices. We will work with files but they are not the only type of I/O devises. For example, network communication relies on the same constructs.

Java I/O

- I/O = Input/Output
  - In this context it is input to and output from programs
  - Input can be e.g. from the keyboard, a file or a socket
  - Output can be e.g. to the display, a file or a socket
- Java I/O is accomplished using *streams of data* – objects that either
  - Deliver data to their destination (screen, file, etc.), or
  - Take data from a source (keyboard, file, etc.)
- There are a very large number of classes in the `java.io` package
  - We will cover the most common ones here

Streams

- You will always need either
  - An input stream when reading data – `java.io.InputStream`
  - An output stream when writing data – `java.io.OutputStream`

- A stream connects a program to an I/O object

- For example, standard I/O streams in Java are:
  - `System.in` – standard input stream that connects a program to the keyboard
  - `System.out` – standard output stream that connects a program to the screen (console)
  - `System.err` – standard error output stream (typically also connected to screen/console)

InputStream and OutputStream

- `InputStream` and `OutputStream` allow you to read and write *raw bytes*
  - `int read()`
  - `int read(byte[] b)`
  - `int read(byte[] b, int off, int len)`

  - `void write(int b)`
  - `void write(byte[] b)`
  - `void write(byte[] b, int off, int len)`
- Both are abstract classes, so a suitable subclass is needed
  - `System.in, FileInputStream, ObjectInputStream`
  - `System.out, FileOutputStream, ObjectOutputStream`

Reader and Writer

- To read or write *16-bit characters*, you will need a `java.io.Reader` or `java.io.Writer` (both abstract)
  - `int read()`
  - `int read(char cbuf[])`
  - `int read(char cbuf[], int offset, int length)`

  - `int write(int c)`
  - `int write(char cbuf[])`
  - `int write(char cbuf[], int offset, int length)`
- Concrete subclasses are, for example:
  - `InputStreamReader, FileReader, BufferedReader`
  - `OutputStreamWriter, PrintWriter, BufferedWriter`

Standard I/O Stream Example

- A simple example on how to read user input from and write output to the console

```
// Read characters from System.in (instead of raw bytes)
InputStreamReader isr = new InputStreamReader(System.in);
// Add buffering capabilities
BufferedReader br = new BufferedReader(isr);

// Output to console using System.out stream
System.out.print("Enter some text: ");

// Input from console, buffered one line at a time
String textEntered = br.readLine();

System.out.println("You wrote :" + textEntered);
```

Reading and Writing Files

- The same principles apply to file streams as do for standard streams

- Classes for file reading:
  - `FileInputStream` – read bytes from a file (usually a binary file)
  - `FileReader` – read 16-bit characters from a text file
  - `BufferedReader` – add buffering capabilities if needed

- Classes for file writing:
  - `FileOutputStream` – write bytes to a file (usually a binary file)
  - `PrintWriter` – write 16-bit characters to a text file
  - `BufferedWriter` – add buffering capabilities if needed

code
tutors

Binary Files Example

Also demonstrating Try-with-resouces and AutoCloseables

```
try (FileOutputStream out = new FileOutputStream("test.dat")){
    byte[] dataOut = {0xF, 0x1};
    out.write(dataOut);
    // out.close();
}
catch (FileNotFoundException fnfex) {  /*TODO*/ }
catch (IOException ioex) {/*TODO*/}

try(FileInputStream in=new FileInputStream("test.dat")){
    int dataIn = in.read();
    while(dataIn != -1) {
        // Handle data and read next byte
        dataIn = in.read();
    }
    // in.close();
}
catch (FileNotFoundException fnfex) {  /*TODO*/ }
catch (IOException ioex) {/*TODO*/}
```

All streams implement AutoCloseable, when try block is left, close is automatically called

You could also use File-class to get the size of the file, create byte-array of same size and read the file with single read-statement

Text Files Example

- Several options here, most straight-forward is demonstrated
- Often, we want to read Strings up to the linefeed
  - BufferedReader may be used for that

```
try(PrintWriter writer = new PrintWriter("C:\\Temp\\test.txt")){
   writer.println("Hello, world!");
}
catch(Exception ex) {/*TODO*/}

try(BufferedReader reader=new BufferedReader(new FileReader("C:\\Temp\\test.txt"))){
    String s=reader.readLine();
}
catch(Exception ex) {/*TODO*/}
```

Closing Streams

- Always close the stream when you are done with it
  - And for output streams also use flush
  - Don’t close System.in, System.out nor System.err
- It is enough to close the ”topmost” decorator
- If you implement a class that uses stream in several methods
  - Supply close-method
  - Implement AutoCloseable
  - Also implement finalize for cases where the user of your class ”forgets” to call close

Path Names in Java

- Typical UNIX path name:
  - /user/timmy/home.work/java/myFile.java
- Typical Windows path name:
  - C:\Java\Programs\myFile.java
- Within a string backslashes must be escaped:
  - “C:\\Java\\Programs\\myFile.java"
- Java will accept path names in UNIX or Windows format
  - Regardless of which operating system it is actually running on

File Class

- Allows you to access various properties of a file
  - Also acts like a wrapper class for file names and paths

- `File` has some very useful methods
  - `exists()` – tests if a file already exists
  - `canRead()` – tests if the OS will let you read a file
  - `canWrite()` – tests if the OS will let you write to a file
  - `delete()` – deletes the file, returns true if successful
  - `length()` – returns the number of bytes in the file
  - `getName()` – returns file name, excluding the preceding path
  - `getPath()` – returns the path name (the full name)

```
File numFile = new File("numbers.txt");
if (numFile.exists()) {
    System.out.println(numFile.length());
}
```

Exercise

- Study java.IO-package
- Create files´-package to your solution and FileTests class there
- Experiment with couple of ways on how to create or read a text file

Exercise

- Create a simple line-editor
- Asks user what to do
  - 1 = Add a new line of text
    - Ask for contents of that line
  - 2 = Edit existing line
    - Ask line number
    - Ask for new contents of that line
  - 3 = Save
    - Ask for name of a text file, save information there
  - 4 = Read
    - Ask for name of a text file, read information
- After each option the lines of text are displayed
  - Line-number + actual text on that line

Serialization

Binary serialization

Serialization case RMI

JAXB

Java Serialization

- Serialization allows you to write objects into stream so that they can be restored back to their original state
- Class must allow serialization by implementing Serializable interface
- ObjectOutputStream and ObjectInputStream are used

```
FileOutputStream serOut = new FileOutputStream("C:/myObj.ser");
ObjectOutputStream oos = new ObjectOutputStream(serOut);
oos.writeObject("Hello");
oos.writeObject(new Date());
oos.close();

FileInputStream serIn = new FileInputStream("C:/myObj.ser");
ObjectInputStream ois = new ObjectInputStream(serIn);
String s = (String)ois.readObject();
Date d=(Date)ois.readObject();
ois.close();
```

XML-serialization with JAXB

- JAXB defines the bindings from XML to Java
  - Use schema to create Java classes
  - Transfer data between objects and XML

```
try{
    JAXBContext ctx=JAXBContext.newInstance(Car.class);
    Marshaller m=ctx.createMarshaller();
    m.marshal(this, new File(“car.xml”));
}
catch(Exception ex){
    ex.printStackTrace();
}


try{
    JAXBContext ctx=JAXBContext.newInstance(Car.class);
    Unmarshaller m=ctx.createUnmarshaller();
    Car c=(Car)m.unmarshal(new File(“car.xml”));
}
catch(Exception ex){
    ex.printStackTrace();
}
```

Many different sources may be used

JAXB

- JAXB-serialization may be used if rules set by JAXB-technology are followed
  - Public class
  - Public default constructor
  - Publicly available data will be serialized (public getter and setter)
  - Class is annotated with @XmlRootElement

We'll work with the following Car

```
@XmlRootElement
public class Car {
	private int id=1;
	private String make="Volvo";
	private Date dateOfCommissioning=new Date();

	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}

	. . . Other getters and setters
}
```

JAXB-Annotations

- @XmlRootElement can also set the name of serialized class
- @XmlType can be used to set order of the fields
- @XmlAttribute in XML-representation moves the field to attribute
- @XmlAttribute and @XmlElement may be used to give name to attribute or element

```
@XmlRootElement(name=“CarObject")
@XmlType(propOrder={"id","make","dateOfCommission"})
public class Car {
    @XmlAttribute(name="carid")
    public int getId() {
        return id;
    }

    @XmlElement(name="date")
    public Date getDateOfCommission() {
        return dateOfCommission;
    }
    . . .
}
```

Exercise (extra)

- Jaxb-serialization is not available in Java SE 11
- With maven it can be enabled

- Google…..

- Try to make JAXB-serialization work for books in the BookApp

- We will come back to this topic as we start implementing server applications

So today

- We covered basics of java.IO-package and you should be able to create and read text files. Possibly you also have some understanding how the streams can be used in wider context.

- Seuraavaksi:
  - Kopioi awtrainer-hakemiston CPHarjoitus1-kansio omaan git-hakemistoosi
    - Omassa hakemistossasi tee
      - git add –A
      - git commit –m “Checkpoint harjoituspohja”
      - git push
- Tämän jälkeen
  - Voit avata CPHarjoitus1:n NetBeans:llä ja toteuttaa siinä olevia harjoituksia
  - Voit työstää BookApp-harjoitusta
  - Voit katsoa itsenäisesti läpi eilisen aineiston Streams-osuutta ja tehdä valikoituja osia sen harjoituksista
- Pysytään virtuaalisessa luokkahuoneessa
  - Mutta kaikilla (myös Jyrkillä) on vapaus silloin tällöin jaloitella pois koneen ääreltä
  - Kuitenkin kello 16:00 kaikki taas koneella viimeistä kertausta varten