TND002 Object-oriented programming

Lecture 6: IO and Java Exceptions

Mark Eric Dieckmann,
MIT division,
ITN, Linköping University.

January 26, 2023

Outline

- IO in Java.
- Exceptions (errors) and how to deal with them.
- Reading from the console.
- Reading from a file.
- Writing to a file.

The IO system

Java's IO system works in the way of a pipeline.

We can read in data and write data to the console and to external files.

Data flow involves "streams" located in the class **System**. Streams are objects that are accessed via a reference variable.

System.in connects an external device to your code.

System.out connects your code to an external device.

The latter is set by default to the console:

System.out.println(arg) writes the content of *arg* to the console.

Java exceptions (=error in Java)

Accessing the console or files can give exceptions.

Exceptions are instances of a class with a name that depends on what the problem is.

There are two types of exceptions in Java:

Unchecked ones cannot be predicted: for example converting the string "abc" into an integer.

Checked ones: The JVM warns you about them. Examples: initializing a variable with a wrong value or IO exceptions.

Action 1: you fix it or inform the JVM that you know that.

Action 2: you use a *try/catch* block to prevent it.

Reading from the console

By default, *System.in* is connected to the console, which provides a stream of int encoded in UTF-8 format.

An **InputStreamReader** (library *java.io*) can access *System.in*

Accessing the console can give checked exceptions:

```
package yourPackageName;
import java.io.*; // we import all classes from the IO library.
public class MainClass {
public static void main(String[] args) throws IOException {
```

We place a *throws IOException* in the method where the exception may occur: Action 1.

This does not fix the problem. The code crashes if an *IOException* occurs.

Reading from the console

This program reads in one UTF-8 encoded int from the console.

```
import java.io.*;
public class MainClass {
public static void main(String[] args) throws IOException {
InputStreamReader consoleReader =
new InputStreamReader(System.in);
System.out.print("Input: "); int in = consoleReader.read();
System.out.println(in);}}
```

System.out.print(arg); does not jump to the next line.

The instance method *read()* of **InputStreamReader** reads in the character you typed in and expresses it as an int.

"a" gives the value 97. UTF-8 is an extended form of ASCII.

Reading from the console

Instances of **BufferedReader** have a method *readLine()* that reads in all characters until we hit return.

It returns a string composed of the characters that correspond to UTF-8 rather than integers.

We can send an instance of **InputStreamReader** into the argument list of the constructor of **BufferedReader**.

```
InputStreamReader i = new InputStreamReader(System.in);
BufferedReader b = new BufferedReader(i);
```

or create an instance directly in the argument list

```
BufferedReader b = new BufferedReader(new
InputStreamReader(System.in));
```

Reading from the console

```
import java.io.*;
public class MainClass {
public static void main(String[] args) throws IOException {
BufferedReader b = new BufferedReader(new
InputStreamReader(System.in));
String in; // Declared outside the loop.
do { System.out.print("Input: "); in = b.readLine();
System.out.println(in); } while (!in.equals("end"));
b.close() ;}} // we close the reader to free up resources.
```

readLine() reads in the string we type in until we press return.

! is the logical negation operator.

The do-while block runs until we type "end".

Catching the Exception

throws sends the exception from the method where it occurred through all calling methods until it reaches the JVM (crash).

Java allows us to catch the exception on its way to the JVM.

```
import java.io.*;
public class IOTest {
private static BufferedReader reader =
new BufferedReader(new InputStreamReader(System.in));

public static void main(String[] args) { // No throws needed
System.out.println(readFromConsole()); }

public static String readFromConsole() { // No throws needed
System.out.print("Type something: ");
try { String s = reader.readLine(); return "You typed: " + s;}
catch(IOException ierr) {return "IO Exception";} }}
```

Exception handling

Try-catch blocks can also catch other exceptions.

```
String s = "Hello"; int i = Integer.parseInt(s); // unchecked one
```

The console error message gives the class of the exception:

```
try { String s = "Hello"; int i = Integer.parseInt(s); }
catch(NumberFormatException ierr)
{ System.out.println(ierr.getMessage()); }
```

ierr is an instance of **NumberFormatException**.

It is a local variable in the catch block and can be used there. Our catch block writes to the console:

For input string: "Hello"

Reading from a file: the class File

An instance of *File* can be connected to an external file.

We can interact with the file through instance methods of **File**:

```
File f = new File("File.txt");
f.canRead(); // boolean: true if we can read from the file.
f.canWrite(); // boolean: true if we can write to the file.
f.exists(); // boolean: true if the file exists.
f.createNewFile(); // boolean: true if it can create the file.
f.delete(); // boolean: true if it can delete the file.
```

File takes the role of **System.in** for files.

You cannot read in text with the instance of **File**. You need a file reader that takes the role of the **InputStreamReader**.

Reading from a file: the class FileReader

```
FileReader fr = new FileReader(arg); // arg is of type File.
FileReader fr2 = new FileReader(filename);
// filename is a String (overloaded constructor).
```

Project is the default directory for files. You can copy a file into it by right-clicking the project name and pasting the file into it.

You can also find the current directory with

```
System.out.println(System.getProperty("user.dir"));
```

You can add what *System.getProperty("user.dir"))* returns to the filename (full path of file) and use that for *arg*:

```
String fullPath = System.getProperty("user.dir") + "\" + arg;
```

Reading from a file: the class BufferedReader

As for console input, we use a **BufferedReader** to convert the UTF-8 code into strings.

```
File file = new File(String arg); // arg is the filename
FileReader fr = new FileReader(file);
BufferedReader br = new BufferedReader(fr);
```

is the long form, which can be shortened into

```
BufferedReader br =
new BufferedReader(new FileReader(arg)); // filename arg
```

We can access one line from the file (until its end) with

```
String text = br.readLine();
```

Reading a sequence of strings from a file

We can read all lines from a file *TextSource.txt* with:

```
public static void main(String[] args) throws IOException {
BufferedReader br =
new BufferedReader(new FileReader("TextSource.txt"));
String result;
while((result = br.readLine())!=null) {
System.out.println(result);}}
```

We can assign the value from *br.readLine()* to *result* in the loop condition and use it in the loop.

If you want to keep the text you can write it into a dynamic array.

```
theArray.add(result); // where theArray was initialized as
ArrayList⟨String⟩ theArray = new ArrayList⟨String⟩();
```

Writing to a file

The easiest way to write to a file is to combine instances of **File**, **FileWriter** and **BufferedWriter**.

We create a new file as

```
File f = new File(fname); f.createNewFile(); // fname is a string
```

We can connect this file to instances of the classes above as

```
BufferedWriter w = new BufferedWriter(new FileWriter(f, b));
```

If the boolean (*b== true*), new text gets appended to the file.

If (*b == false*), the file content gets overwritten.

Writing to a file

Our buffered writer *w2* overwrites the file attached to *f*.

```
Writer w2 = new BufferedWriter(new FileWriter(f, false));
w2.write("Hello\r\n"); // \r\n needed for new line in a file.
w2.write("world"); w2.newLine(); // also creates a new line.
w2.flush(); // we need to flush the buffer.
```

Text we send into *w2.write(..)* is stored in its buffer until the buffer is full. Then the entire buffer is copied into the file.

If we do not flush it, the content remains in the buffer.

Closing a BufferedWriter flushes it.

Summary

We discussed

- IO in Java.
- Exceptions (errors) and how to deal with them.
- Reading from the console.
- Reading from a file.
- Writing to a file.