Introductory Programming
Exceptions and I/O: sections 8.0 – 8.3

Anne Haxthausen[a]

IMM, DTU

1. Exceptions (section 8.0)
2. Input and output (I/O) (sections 8.1-8.3)

a. Parts of this material are inspired by/originate from a course at ITU developed by Niels Hallenberg and Peter Sestoft on the basis of a course at KVL developed by Morten Larsen and Peter Sestoft. Translated into English by Philip Heede.

Errors in programs

Three types of errors:

1. Syntax errors: detected by the `javac` compiler
2. Runtime errors (e.g. divide by zero): detected by the `java` interpreter and gives rise to exceptions
3. Logical errors (the program does something else than planned): possibly discovered by testing

An *exception* is an event that occurs, when an error is detected during the execution of a program. It causes execution to be halted, unless you perform *exception handling*.

Exceptions

Programs must be able to signal and handle error situations — by throwing and handling *exceptions*.

- What is an ‘exception’?
- How do you throw (Danish: kaster) an exception (using a **throw** statement)?
- How do you handle(Danish: håndterer) an exception (usning **try-catch** clauses)?
- When do you declare an exception (using **throws**) in the method header?
- User-defined exception classes.

What is an exception?

An exception is an instance of a subclass of the class `Throwable`.

It is used to signal and describe an error condition, or another *exception*al situation, that occurs at runtime.

The object includes:

- what exception class it is an instance of
- a description of the cause of the exception
- information about where the exception occurred

This information is printed by a call to the object’s `printStackTrace` method. The object’s `getMessage` method returns the message as a string.

Here is part of the class hierarchy for `Throwable`, `Error`, and `Exception`:

```
Throwable
    Error                               (need not be declared)
        VirtualMachineError
            OutOfMemoryError
            StackOverflowError
    Exception                           (must be declared)
        IOException
            FileNotFoundException
        RuntimeException                (need not be declared)
            ArithmeticException
            IndexOutOfBoundsException
            IllegalArgumentException
                NumberFormatException
```

Throwing of exceptions

An exception is *thrown* when a runtime error occurs during the execution of a program.

Two types of runtime exceptions:

- pre-defined: a runtime exception is thrown automatically by the Java interpreter when an error is detected
- user-defined: an exception is explicitly thrown by the programmer using the **throw** statement

Example of an automatic (pre-defined) exception

When running

```
public class ArrayIndex  {
  public static void main (String[] args) {
          int[] a = {1,2,3,4,5};
          System.out.println(a[5]); // indexing error
          System.out.println("Will not print");
  }
}
```

the program is aborted with an error message:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
    at ArrayIndex.main(ArrayIndex.java:4)
```

Example of explicitly thrown exception

The Date constructor below allows the construction of 'illegal' dates, e.g. 1999-13-30 is constructed with `Date(1999, 13, 30)`

```
Date(int year, int month, int day) {
  this.year = year; this.month = month; this.day = day; }
```

A better solution: the constructor throws an exception, if it is asked to construct an illegal date:

```
class Date {
  ...
  Date(int year, int month, int day) throws Exception {
    if (ok(year, month, day))
      { this.year = year; this.month = month; this.day = day; }
    else
      throw new Exception("Illegal date: "
                          + year + " " + month + " " + day);
  }
...
}
```

When running

```
public class Date-exn01 {
    public static void main(String[] args) throws Exception {
      Date d1 = new Date(1999, 13, 30); //illegal date
      System.out.println(d1);
    }
}
```

the program is interrupted with an error message:

```
Exception in thread "main" java.lang.Exception: Illegal date: 1999 13 30
    at Date.<init>(Date.java:8)
    at Date-exn01.main(Date-exn01.java:3)
```

Exception handling

Ways to handle an exception:

1. ignore them as in the previous example
2. catch them (with **try-catch**)
   (a) where it is created
   (b) somewhere else in the program

Solution 1 can give excellent error descriptions but will cause the program to exit in the middle of operations, which can be unfortunate.

Solution 2 gives the possibility that the program won't crash on an error, but can be 'saved' by taking special action.

If solution 2b is used, you can collect errors from multiple parts of your program in one location — good, if you are interested in catching errors for part of the program but you don't care about exactly where the error occurred.

Ad 1: unhandled exceptions

When an exception is ignored in the program, the following happens:

1. the program stops
2. a message is printed with information about
   (a) what exception occurred
   (b) where in the program it occurred
      - the first line contains the name of the method in which the exception was thrown and any message that may be attached
      - the remaining lines show which methods were called to reach the point of the failure

Ad 2: catching exceptions with try-catch clauses

```
try
{
  statements1
}
catch (Exception exn)
{
  statements2
}
```

- The statements in *statements1* are executed.
- If an exception `e1` is thrown during the execution of *statements1*, the statements in *statements2* are executed. Otherwise the statements in *statements2* are ignored.
- During execution of *statements2*, the variable `exn` is bound to the exception object thrown during the execution of *statements1*.

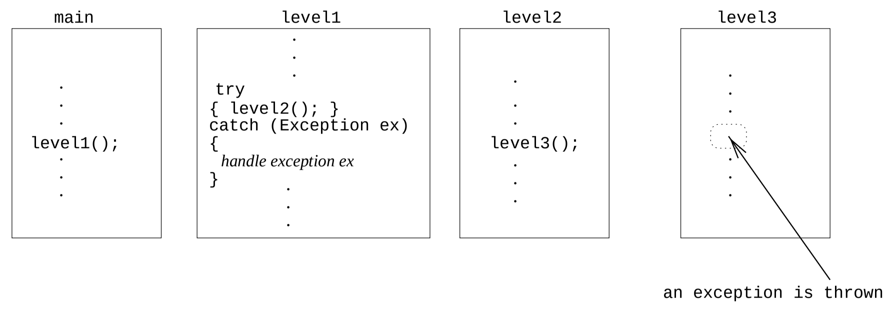
Ad 2: example of catching an exception

```
public class Date-exn1 {
  public static void main(String[] args) {
    try {
      Date d1 = new Date(1999, 13, 30);
      System.out.println(d1);
    }
    catch (Exception e)
    {  System.out.println("Illegal date!"); }
    System.out.println("We'll go to here!");
  }
}
```

gives the following output:

```
Illegal date!
We'll go to here!
```

Ad 2b: exception propagation



Declaring exceptions

The exceptions `Error` and `RuntimeException` and their sub-classes are 'unchecked'.
All other exceptions are 'checked'.

For 'checked' exceptions the following rules apply:

If for a method `m`, there is a risk that an exception `E` is thrown when it is called with `m(...)` and that this exception is not caught in the body of the method, then it must be declared in the header of the method with **throws** `E`.

Example: see `Date-exn01`

'Unchecked' exceptions are never declared.

User-defined exception classes

It is possible to create your own exception classes.

For an example: see listing 8.6 in the book.

I/O in Java

- Streams
- Input from the keyboard and output to the screen
- Input from and output to text files

Streams

In Java you use *input* and *output* streams to communicate with the surroundings: files, the keyboard, etc.

Java has a family of classes to create stream objects. See figure 8.3 in the book.

Streams can be divided into:

- input streams that have methods to read data from the surroundings
- output streams that have methods to write data to the surroundings

and into

- character (text) streams that interpret the individual bytes as text characters
- byte (binary) streams, where bytes are not interpreted but used 'raw'

If anything goes wrong when performing I/O an `IOException`, which is a sub-class of `Exception`, is thrown.

Reading from the keyboard and writing to the screen

We already know how to:

- *read from the keyboard using the methods in the* `Keyboard` *class*
- *write to the screen using the methods* `System.out.print` *and* `System.out.println`

Example:

```
public static void main(String[] args)  {
  int x;
  System.out.print("Enter a number: ");
  x = Keyboard.readInt();
  System.out.println("The number is: " + x);
}
```

```
Enter a number: 13
The number is: 13
```

Text files vs. binary files

A file is a named collection of data stored on a secondary storage device (e.g. hard disk, floppy disk or cd).

All files consist of bytes. 1 byte = 8 bits. A bit is either `0` or `1`.

It is, however, different how bits are interpreted in different situations.

Example:

A file with four bytes:

01001010	01100001	01110110	01100001

If it is a **text file**, where each byte is to be interpreted as an ASCII character, the four bytes represent the following characters:

J	a	v	a

But if it is a **binary** file, the four bytes can be interpreted as something completely different, e.g. a single integer (of type `int`).

Text files

A text file is a sequence of characters.

Typical filenames: `score.txt, numbers.txt, Time.java,` . . .

Example: The text file `score1.txt` that looks like this

```
Joe 3 4
John 4 5 6
```

consists of the following characters:

J	o	e		3		4	\n	J	o	h	n		4		5		6	\n

Idea when reading text files

When reading text files, it is often more interesting to read whole words and numbers rather than the single characters.

Often we don't care about a specific group of characters jointly called 'whitespace': blank, new line and tabulator.

The individual numbers and words are called *tokens*.

Whitespace often separates the individual tokens.

The first line in `score1.txt` has 3 tokens:

Joe	3	4

The second line in `score1.txt` has 4 tokens:

John	4	5	6

Reading of text files, step by step

1. Opening the file `score1.txt` and conversion to a buffered reader:
   ```
   FileReader r = new FileReader("score1.txt");
   BufferedReader in = new BufferedReader(r);
   ```
2. You can now read one line at a time from "score1.txt" using `in.readline()` that returns a `String`:
   ```
   String line = in.readline();
   ```
3. Reading a single line token by token:
   ```
   StringTokenizer tokenizer = new StringTokenizer(line);
   String token = tokenizer.nextToken();
   ```
4. Convert `token` to an appropriate type: e.g. to an `int` using
   ```
   Integer.parseInt(token);
   ```

The whole process is wrapped in a **try-catch** clause to catch any exceptions that may occur.

Example of reading a text file

Assignment:

Each line in the file `score1.txt` consists of the name of a golf-player and the number of strokes used for each of the holes.

These data are to be read; and for each player the name of the player, the number of holes played, and the total number of strokes used, is to be output on the screen:

```
Joe has played 2 holes and used 7 strokes
John has played 3 holes and used 15 strokes
```

Solution: see `Reading.java` which can be downloaded from the homepage with the overheads.

Writing to a text file

```
public static void main(String[] args) throws IOException {

  // setup output stream with name "nn" (e.g. "res.txt")
  FileWriter wri = new FileWriter("nn");
  PrintWriter out = new PrintWriter(wri);

  //output to the file "nn" using the methods
  //‘out.print’ and ‘out.println’
  ...

  out.close(); // close the file after use
}
```

`out` above works the same way as our old friend `System.out` — i.e. we can, among other things, write `out.println("Eric");` or `out.print(42);`.

try-catch could also have been used here – then **throws** ... could have been avoided.