CS1812 Lab 2

Goals

In this lab you will play with exceptions. You will catch them, throw them, and you will also create a custom exception.

Before you start

- ☞ Create a directory named `Lab2` in the `OOPlabs` directory where you will keep all the programs from this Lab.
- ☞ Download the archive `lab-2-assets.zip` from the CS1812 Moodle page to the `Lab2` directory and unzip it. You should have the following files in your directory:
  - `BadReadInt.java`
  - `CatFile.java`
  - `Factorial.java`
  - `ReadInt.java`
  - `Z.java`

  You may now want to delete the archive with the command `rm lab-2-assets.zip`.
- ☞ This labsheet consists of a set of exercises, which are grouped into four mandatory checkpoints. You are expected to work on solutions to the exercises before and during the actual lab session.
- ☞ The checkpoints will be verified by a TA during the lab session. You can also discuss improvements to your solutions, or ask questions in case you are having problems.
- ☞ At the end of the labsheet, you will find a section called *Going further* corresponding to a Platinum Checkpoint. Those exercises are optional, but if you finish the other (mandatory) checkpoints, try to complete those exercises as well.
- ☞ Make sure you preserve the indentation (using the tabulation) when typing your programs. You will not get checkpoints verified unless your code is properly formatted.
- ⚠ **Make sure you have finished all the mandatory checkpoints from the previous lab before starting this one.**

1 Catching exceptions

"When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. (...) After a method throws an exception, the runtime system attempts to find something to handle it. (...) The exception handler chosen is said to catch the exception."

Java Oracle Tutorial: What Is an Exception?

1.1 Input validation

In this week's assets you will find the file `BadReadInt.java`. The contents of the file are as follows.

```
import java.util.Scanner;

class BadReadInt {
    public static void main(String[] args) {
        int n;
        Scanner s = new Scanner(System.in);
        System.out.print("Enter an integer: ");
        n = s.nextInt();
        System.out.println("Good job!");
    }
}
```

Compile the program and run it.

☞ Enter a string instead of an integer.

The runtime environment will throw an exception of the class `InputMismatchException`. Let us catch it: in the file `ReadInt.java` you will find the skeleton of a program that does so.

```
import java.util.Scanner;

    public static void main(String[] args) {
        boolean gotInt = false;
        int n;
        Scanner s = new Scanner(System.in);
        do {
            try {
                //...
                gotInt = true;
            }
            catch(InputMismatchException e) {
                e.printStackTrace();
                System.out.println("The input has to be an integer!");
            }
        } while (!gotInt);
    }
}
```

Compile the program. The compilation will fail, as the class `InputMismatchException` has not been imported.

- Import the class `InputMismatchException` from the `java.util` package.

Now the program will compile. Don't run it yet, as you will have to add some stuff before it works as intended. The behaviour should be as follows.

- The program loops until the user inputs an integer. This is done via a **do**–**while** loop that is controlled by the boolean `gotInt`.
- The **try** block is where the values are scanned from the command line. If the statement `s.nextInt()` throws an exception, then that statement and the rest of the block are not executed. Instead, the program flow moves to the **catch** block where the exception is handled.

Let us have a closer look at the **catch** block. Its argument is an object `e` of the class `InputMismatchException` (the name of the object, `e`, is just a variable name and, as such, is arbitrary). All **catch** statements are of the form **catch**(`ExceptionType name`).

All exceptions are subclasses of the class `Exception` and one may always catch an exception of that type instead, although the specificity of the exception will be lost. To reinforce that, let us do the following experiment.

- Replace **catch**(`InputMismatchException e`) by **catch**(`Exception e`).

Observe that the program still compiles. This means, in particular, that you won't have to import the class `Exception` when you use it in a program, as it is available in the `java.lang` package that is imported by default and provides all the Java fundamental classes. It also means that the method `e.printStackTrace()` is available in all the exception types: it prints to the command line what happened when the exception was thrown and where it has occured.

Exercise 1. You will complete the `ReadInt` class so that it has the intended behaviour.

a) Start by completing the **try** block. To do that replace the dots by the code that was responsible for the exception thrown in the `BadReadInt` class.

b) Compile and run the program. Input a string instead of an integer and observe that, although the exception is caught, the program loops forever. This happens because the `s.nextInt()` is never completed and, as such, the scanner is stuck on your first (bad) input.

c) Modify the **catch** block so that it gets rid of the bad input (Hint: there is a method in the `Scanner` class that always reads the input as a string.)

d) Compile and run the program for the two following **catch** statements:
   - **catch**(`InputMismatchException e`)
   - **catch**(`Exception e`)

   What differences can you observe?

☑ Checkpoint 1

1.2 Catching two exceptions

Sometimes a **try** block may throw several different exceptions. In that case, we can have a **catch** block for each one of those exceptions.

Exercise 2. Consider the class `Factorial` from this week's assets, corresponding to the recursive factorial that you have seen in a previous lab session.

```
class Factorial {
    private static long fact( long n ) {
        if ( n==0)
            return 1;
        else
            return n*fact(n-1);
    }

    public static void main (String[] args) {
        long a = Integer.parseInt(args[0]);
        System.out.println( fact(a) );
    }
}
```

Compile the program and check that it works. Don't forget that it gets its argument from the command line and, as such, to compute the factorial of 5 you will need to issue the command `java Factorial 5`.

a) If you run the program without any arguments you will get a message with an exception `ArrayIndexOutOfBoundsException`. Modify the program to catch that exception and print a warning when it does. (Hint: add a **try** block and a **catch** block to the `main` method.)

b) Now run the program by issuing the command `java Factorial abc`. You will raise another exception. Which one?

c) Modify the program so that it now catches both exceptions. (Hint: you will need to add another **catch** block to the `main` method.)

d) Like every object in Java, exceptions have a `toString()` method. Add to each **catch** block a line that prints the exception. Compile the program and run it twice, throwing a different exception in each of the executions.

e) Add, after the two other **catch** blocks, a third **catch** block that starts with **catch**(`Exception e`). Compile the program and observe that everything goes well. Then, place that new **catch** block *before* the other two. Why does the compilation fail now?

f) Comment out the other two **catch** blocks so that the program compiles with the **catch** block from the previous question – the one that starts with **catch**(`Exception e`). What do we lose with this modification?

☑ Checkpoint 2

1.3 The finally block

"The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling – it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated."

Java Oracle Tutorial: The finally Block

In the following exercise you will play with a **finally** block and understand its intricacies.

Exercise 3. Consider the following class from this week's assets (file `Z.java`).

```
public class Z
{
    private static String silly() {
        try
        {
            int n = 1/0;
            return "A";
        }
        catch (Exception ex)
        {
            System.out.println("B");
        }
        finally
        {
            System.out.println("C");
        }
        return "D";
    }

    public static void main(String[] args)
    {
        System.out.println( silly() );
    }
}
```

Compile the program and run it.

a) Explain the output of the program.
b) Remove the following line from the `silly` method:
   `int n = 1/0;`
   Compile the program and run it. Explain the new output.
c) Remove the **finally** block from the program and repeat the two previous questions.

☑ Checkpoint 3

2 Throwing exceptions

"Before you can catch an exception, some code somewhere must throw one. (...) Regardless of what throws the exception, it's always thrown with the throw statement."

Java Oracle Tutorial: How to Throw Exceptions

2.1 The throw statement

The syntax of a throw statement is the following.

```
class Test {
    public void myMethod() throws SomeException {
        ...
        throw new SomeException();
        ...
    }
}
```

When the program flow reaches the **throw** statement, the exception `SomeException` is thrown. It can be caught in some other part of the code via the statement **catch**(`SomeException e`).

2.2 Custom exceptions

To define a custom exception, you have to create a new subclass of the class `Exception`. In a future lab session we will deal with inheritance and understand in detail the creation of a subclass. But, for now, an empty subclass will do.

Therefore, you can use the following syntax to create a custom exception called `MyException` and throw it in a `Test` class.

```
class MyException extends Exception {
}

class Test {
    public void myMethod() throws MyException {
        ...
        throw new MyException();
        ...
    }
}
```

2.3 Putting it all together

In the following exercise you will create a custom exception, throw it and catch it.

Exercise 4. In the context of Exercise 2 do the following.

a) Create a class corresponding to a custom exception called `NegativeInteger`.

b) Change the signature of the `fact` method in the `Factorial` class, and have it throw the exception `NegativeInteger` when the user inputs a negative integer. Can you understand why you get an error when you try to compile the program?

c) In the `main` method of the `Factorial` class, write a **catch** block that catches the custom exception.

d) Finally, compile the program and run it with an input that throws the custom exception.

☑ Checkpoint 4

3 Going further

Exercise 5. In the context of Exercises 2 and 4, modify the program so that it asks the user for the input instead of reading it from the command line arguments. Then, each time the user input throws an exception, ask the user to enter another input repeatedly until no exception is thrown.

Exercise 6. In the assets there is a file `CatFile.java` with the following contents.

```
import java.io.*;

class CatFile {
    public static void cat(File file) {
        RandomAccessFile  input = null;
        String line = null;
        try {
            input = new RandomAccessFile(file, "r");
            while ((line = input.readLine()) != null) {
                System.out.println(line);
            }
            return;
        }
        finally {
            if (input != null) {
                input.close();
            }
        }
    }
}
```

Modify the program so that it compiles.

☑ Platinum Checkpoint