

This lab concerns exception handling. It is optional, and you do not have to show it to your TA, but the material *is* on the final. This Lab will be easier if you did the reading on exception handling, Chapter 10 of the course text.

Download file `Lab12.java` from the course web page. Put it in a new directory, open the file in DrJava, and compile.

**Task 1. Get familiar with function `getKeyboard`.** You will be using this function later. Read its specification and try it out by typing this into the Interactions pane.

```
c= Lab12.getKeyboard();
c.readLine()
```

When the box appears in the Interactions pane, type something into it and hit the return/enter key.

**Task 2. Get familiar with checked exceptions.** Study function `readKeyboardLine`; make sure you understand the specification and the function body.

Note the throws-clause `throws IOException` in the function header. *IO* stands for *input-output*, and an `IOException` is thrown whenever some sort of input-output error happens.

Delete the clause `throws IOException` from the header of function `readKeyboardLine` and try to compile. You get an error message from the compiler, right? Read it. Then put the throws-clause back in the function header and compile.

As explained on pp. 323-325 of the text, Java expects that an exception that may be thrown in a method body either (1) be caught in that method body or (2) be mentioned in the throws clause of the method header, so that the user has a syntactic indication that the exception may be thrown. All exceptions that may be thrown must be checked except for instances of `RuntimeException` (and its subclasses).

Do not actively be concerned about these throws clause. Instead, work as follows. Whenever you try to compile but Java says that a throws clause is needed, put it in.

**Task 3. Complete function `readKeyboardInt`.** From the spec of this function, you know that it is supposed to keep prompting the user for an integer, giving them a message when they type something different from an integer. For this purpose, use an infinite loop whose body is a try statement:

```
try {
    ...
} catch (decl. of an exception variable) {
    ...
}
```

where the following describes the try- and catch-blocks:

1. The try-block reads a line from the keyboard and returns the value of a suitable call on function `Integer.parseInt`. To help you out, the spec for `Integer.parseInt` is given to the right.
2. The catch-block catches any exception thrown by `Integer.parseInt` and prints (using `System.out.println`) a suitable message for the user. Note that the declaration of an exception parameter, just like a parameter of a method, is simply a variable declaration (of the form “*type-or-class variable-name*”) that appears inside the parentheses after `catch`. In this case, the type-or-class is the class of the exception thrown by `Integer.parseInt`.

**Task 4. Using exception handling for preconditions to aid in robust programming.** Throughout this course, we have used the term *precondition* for a true-false statement that should be true when the method is called, and it is the caller’s duty to ensure that it is true. Take a look at function `exp` in class `Lab13`. It has the precondition  $c \geq 0$ . In this case, if the user calls `exp` with  $c < 0$ , infinite recursion results.

**From the API specification for `Integer.parseInt` (and then edited)**

```
public static int parseInt
    (String s)
    throws NumberFormatException
```

Parse *s* as a signed decimal integer. The chars in *s* must be decimal digits, except that the first character may be an ASCII minus sign ‘-’ (“\u002D”) to indicate a negative value. Return the resulting integer value. Throw a `NumberFormatException` if *s* does not contain a parseable **int**.

Exception handling can be used to enable more robust programming. A robust program will prevent abnormal termination and unexpected actions—like this infinite loop. Robust programming requires that invalid inputs, for example invalid arguments of a call, be handled in a reasonable way. In the case of function `exp`, the function itself doesn't know how to handle the error  $c < 0$ , but it can throw an exception so that the user can handle it.

Change the specification of function `exp` so that it says that it throws an exception if  $c < 0$ —indicating which one is thrown. You can choose an appropriate one from the list given to the right; these (and many more) are defined in the Java API package. Then, change the function itself to throw that exception, with a suitable message in the exception, if  $c < 0$ .

#### Some Java Exception classes

```
ApplicationException
ArithmeticException
ArrayStoreException
FileNotFoundException
IndexOutOfBoundsException
IllegalArgumentException
IllegalStateException
InvalidOperationException
InvalidParameterException
```

Make sure function `exp` works correctly by using the function call `Lab12.exp(1, -3)` in the interactions pane. Write down what happens here:

**Task 5. Writing a throwable class definition.** Define subclass `MyException` of class `Exception`. It needs two constructors: one with no parameters and one with one `String` parameter, which is the detail message. Be sure to write specifications for the constructors.

**Task 6. Writing a procedure to use what you have done.** Suppose  $0 < x < 1$ . How fast does  $x^i$  approach 0 as  $i$  increases? Write function `approach` to find out—its specification is given in class `Lab12`.

Note that for  $x$  satisfying  $0 < x < 1$ ,  $x^0 = 1$ , so that there is some positive integer  $i$  that satisfies the spec.

After testing your function to be sure that it works, try it in the interactions pane with various values of  $x$ , like `.01`, `.99`, `.999`, `.9999`, `.99999`—and of course `0`, `-2`, `1`, and `2` to be sure that throwing an exception works properly. When done, you might want to change the spec and function body to throw an `IllegalArgumentException` instead of a `MyException`. In this case, it makes more sense to use it. You can then delete the “`throws MyException`” clause from the method header.

Show what you have done to your TA or a consultant—let them look at your computer monitor to make sure you wrote the methods correctly.