Bullet-Proof Programs or Input Validation

Due date: by the end of the recitation session

Introduction

Popular Murphy's Law states: "***whatever can go wrong, will go wrong***".

In the previous programming courses, you probably often heard the phrase: "assume a well-behaved user". If we can assume that the input data that comes from the user or files is correct, then the programs become simpler. But the users and input files are generally not well-behaved. At some point we need to be able to write code that can handle real users and real data (i.e., whatever is thrown at it).
What do I mean by *handle*?
The program should not crash with an exception and it should not produce results that make absolutely no sense. Instead, the code should validate any input data and produce appropriate error messages indicating invalid data types or values.

- In some situations the program may be able to request the user to re-enter incorrectly provided information (this makes sense in interactive programs).
- In some cases, the program may need to skip invalid data (and optionally log that fact in an error log file).
- In some cases, the program may have to terminate after producing the error indicating what went wrong.

Not validating data properly may lead to very costly consequences for the users and the companies that use such software. Here is one story from a few years ago that may give you goosebumps: *The $100,000 Keying Error* http://ieeexplore.ieee.org/document/4488265/

Complete the exercises described on the next few pages. Their goal is to give you practice on recognizing the potential for trouble and on bullet-proofing your code (or at least attempting to do so).

Worksheet: **https://goo.gl/PQfYpq**

Part 1

Consider the following short program:

```
import java.util.Scanner;

public class Age {
	public static void main(String[] args) {
		Scanner in = new Scanner(System.in);
		System.out.println("Enter your age (a positive integer): ");
		int num = in.nextInt();
		System.out.println("Enter your name: ");
		String name = in.next();
		System.out.println(name + " is  " + num + " years old." );
		in.close();
	}
}
```

There are more possible inputs that cause problems than there are those that will produce correct and reasonable results.
For each of the input values below state what happens. If the program crashes, find out what exception was thrown. If the program produces unreasonable results, state why the results make no sense. If the program does not encounter any problems, just indicate that. For each input below the first line indicated what the user enters on the first prompt and the second line indicates what the user enters on the second prompt.

1. 21
   Jane
2. -35
   Mark
3. 215
   Joe
4. 3
   R2d2
5. 18
   Ann Marie
6. 111111111111111
   Ann
7. 32.5
   Amber
8. Eighteen
   Steven
9. 32 43 12
   Ellen

Part 2

Handling problems like the ones you observed in Part 1 requires different types of validation techniques:

- If the input data supposed to fall within a certain range (dictated by the specification, instructions or, sometimes, common sense), then the program should make sure that those ranges are met.
- If certain values makes sense and others do not (this should really be determined by the specification), then the program should attempt to detect and reject the values that do not make sense.
- If the program makes calls to functions that throw exceptions, the code should be ready to handle those exceptions (the language like Java forces the programmers to handle the checked exceptions, but it is a good idea to watch out for the unchecked exceptions as well). Catching and handling specific types of exceptions allows use to write appropriate types of handlers for the errors that may occur.
- If the input data comes from the user, the input buffers should be emptied when the input statement is satisfied. In some cases, it might be worthwhile to warn the user (especially in the interactive programs that deal with sensitive information) if part of the input is being ignored.

Rewrite the code from Part 1 so that it handles all of the problems that occurred for inputs specified in Part 1. Then rerun your program and make sure that it does handle all of the errors.

Part 3

Consider the following short program:

```
public class Add {

	public static void main(String[] args) {

		int num1 = Integer.parseInt( args[0] );
		int num2 = Integer.parseInt( args[1] );

		System.out.printf( "Sum of %d and %d is %d. \n",
				num1, num2, num1+num2);
	}
}
```

It is a very different type of program than the one in Part 1. It also faces other types of problems. It works with command line arguments rather than with user input coming in through Scanner object. (Remember that in Eclipse you can specify the command line arguments using *Run > Run Configurations > Arguments tab*.)

- Identify all potential problems (or as many as you can find). Group them into types of problems that are similar in nature (for example, in part 1, entering 32.5 for age caused a problem; entering 2.9 would have caused the same type of problem, so this indicates the need to handle floating point numbers when integers are expected).
- Rewrite the program to handle the errors that your group identified. (Do not prompt the user for alternative inputs, just produce an error message and terminate when any problem occurs.) Make sure that you code never terminates/crashes with an exception. Try to make sure that the code does not produce unreasonable results.

Part 4

Last week you were looking at the source code of OpenMRS package. Let's go there again and see some examples of real life input data validation. Go to OpenMRS main GitHub page at https://github.com/openmrs , select openmrs-core and then directories: `api > src > main > java/org/openmrs > validator` (the worksheet has the direct link to that folder). This directory contains many different classes that provide validation of various types of values obtained from the user or from other external sources. Take a look at `PersonNameValidator.java` class. Read through the code - you may not be able to understand every single line of the code, but you should be able get a rough idea what it is doing. Describe in your own words what makes a valid name in OpenMRS.