7. Reading in Data Safely

***Note**: some example programs are from, or based on, examples from Java for Everyone (C Horstmann), the course text.*

So far we've used the Scanner library to read in strings and integers, and we've observed that if you try to read in an integer and the user types something that is *not* one, then your program will crash. We really need to do something about that. Fortunately, there are several ways to do this.

- **Catch the Error** – we can 'trap' the error and fix it. Java includes features to do this, called *exception handling,* based on the keywords try and catch (essentially you 'try' something and if it goes wrong you 'catch' the error). Exceptions and handling them is part of the CS-115 module and **I would prefer you did not use this technique in this module.** Sometimes tutors and lab demonstrators will suggest it – but it's an advanced technique, that's easy to get wrong (and people commonly do in attempting coursework). It's also easy to write code that is bad practice – for example, having far too much code in the 'trying' part; or doing nothing in the 'catching' part, both of which are not good. And it's longer as well. Having said that, if you use exception handling in the coursework and get it right – that's fine with me. *But if you get it wrong, you will lose marks.*
- **Check for the Error First** – the preferred (in *this* module) way to do it is to check first to see if there's going to be a problem – Scanner provides ways to 'sneak a look ahead' and see what the user's typed before we actually read it in. **I would much prefer you do it this way in this module** – it is shorter and does not rely on you using features of the language we are not doing yet.

We'll illustrate the technique with integers – but you can also do it with other types of data too. The key to the approach is a method that forms part of the Scanner library called `hasNextInt()`. This does the following.
If the user hasn't typed any input yet, it waits.
Once they do type something, it checks to see if it's an integer.
If it is, it returns `true`; otherwise it returns `false`
We can use the `true/false` result from this to decide if we really want to read in the value or not – `hasNextInt()` doesn't actually read it in; it just checks to see if it's the 'right' type. Here's a program based on this:

```
import java.util.Scanner;

class InputInteger {
    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);
        int input;

        System.out.print("Enter an integer: ");
```

```
        while(!in.hasNextInt()){
            System.out.println("Try again.");
            in.next(); //'Throw away' wrong input
        }
        input = in.nextInt(); //actually read the integer
        System.out.println("Thank you. You typed: " +
            input);
    }
}
```

We use the while loop to check that the user has typed an integer:

```
while(!in.hasNextInt()){
```

Is true as long as the input is *not* an integer – because '!' means *not.* So the body of the loop needs to:

- Tell the user they have typed in the wrong thing.
- 'Throw away' the 'wrong' input – remember we haven't read the input yet – just checked what it is. So the input is still sitting there, 'unread' The line in.next(); reads it in but doesn't save it anywhere – it just discards it.

This while loop continues as long as the input is not an integer – it will only finish if the input (waiting, unread as yet) *is* an integer. At this point, it's safe to read it with:

This works fine for integers – but what about other data, like doubles, or Boolean? There are also methods to check to see if the data waiting to be read is a double (`hasNextDouble()`) or a Boolean (`hasNextBoolean()`) – and methods to actually go and read in doubles and Booleans (`nextDouble()` and `nextBoolean()`) once you've checked.

Reading and Checking Specific Strings

Because any string of characters is a String, there's no way our programs can crash if we type the wrong thing. But we still often what to 'force' users to type one of a specific set of strings – for example "yes" or "no". We saw code to do this in the Chapter 6 on Loops when we talked about Do Loops. However, it bears repeating because it's important. You should use code like this:

```
String input;
do {
     System.out.print("Enter either yes or no.");
     input = in.nextLine();
} while (!input.equalsIgnoreCase("yes") &
         !input.equalsIgnoreCase("no"));
```

(Or, better, the version that used final variables for "yes" and "no").

Of course, just reading “yes” or “no” isn’t very useful – you generally want to either do something, or not, depending on what the user typed. The most common ‘pattern’ for code like this is two loops – one inside the other (*nested loops*). Here’s an example

```
//Loop that does 'something'
String input;
do {
     //Code that does the 'something' left out

     System.out.println("Do you want to continue?");
     do {
          System.out.print("Enter either yes or no.");
          input = in.nextLine();
     } while (!input.equalsIgnoreCase("yes") &
               !input.equalsIgnoreCase("no"));
} while (input.equalsIgnoreCase("yes"));
```

The inner loop checks to see if the user types “yes” or “no”; the outer loop carries on as long as they type “yes”.

SHODDY SHORTCUT – DON’T DO THIS

When writing code like the loop above, it’s very common in coursework to leave out the inner loop and just write:

```
String input;
do {
     //Code that does the 'something' left out

     System.out.println("Do you want to continue?");
     System.out.print("Enter either yes or no.");
     //Inner loop left out!
} while (input.equalsIgnoreCase("yes"));
```

The problem with this is the code will assume *anything* other than “yes” means “no” – *including* mistakes the user might make while trying to type “yes!”. So *don’t do it.*