EXCEPTIONS AND INPUT VALIDATION

"The young man knows the rules, but the old man knows the exceptions." [1]
- Oliver Wendell Holmes Sr.

Lab objectives.

In this lab you will learn to handle input and output in your program, with an emphasis on the string as a tool. In the first part of the lab you will learn about input and output via the console. In the second part, you will learn about exception handling and how to validate input to ensure the robustness of your program. Through these exercises that require user input manipulation, you will become aware of the importance of input validation.

Introduction: input and output using the console

The console allows the program to print text output to and receive text input from the user. These two functionalities (output and input), are done using different Java functions.

You actually have already seen the output operator in the earlier parts of the course. The "Hello World!" program uses `System.out.println("Hello World!")` to insert messages to the console. In general, when the programmer wants to print text to the console, they use `System.out.print(String str)` or `System.out.println(String str)`. The difference between those two functions is that `println` will add a new line to the end of the printed line, whereas print will not. The use of `println` makes printing cleaner if you are printing a complete idea, but if you are planning to continue the line, then you can use `print`. For example, in order to print the line:

```
Hey, are you there?
```

Your corresponding code would be:

```
System.out.println("Hey, are you there?");
```

Now it is time for you to write a program that will accept user input, and respond accordingly.

To get user input from the console, it is best to use the `Scanner` class. This class takes the parameter `System.in` (just like `System.out`, this refers to the console) and provides methods for reading the line of input that the user types into the console. To use the `Scanner`, make an instance of the `Scanner` class that takes `System.in` as its constructor parameter.

```
Scanner in = new Scanner(System.in);
```

To read user input, prompt the user to type her input into the console (this is the perfect time to use `print` instead of `println`, since you usually want the user to enter information right after your prompt, not on a new line). Then use the `next`() method on the `Scanner` object to read the line.

```
    System.out.print("Enter your input: ");
    String s = in.next();
```

It is important to note that `in.next()` returns a String object of input to be used by the program. Say you use `.next()` but want to use the input as something other than a String. This seems reasonable. For example, say you have a program that asks the user to input his or her age:

```
import java.util.Scanner;

public class AgeEnterer {
   public static void main(String args[]) {
      Scanner in = new Scanner(System.in);

      System.out.print("Enter your age: ");
      String input = in.next();
      int age = Integer.parseInt(input);
      System.out.println("You are " + age + " years old.");
   }
}
```

If the user enters a correct age, then this will work great! For example, what will print to the console if he enters 18?

But what happens if the user enters an invalid input, i.e. a number that is not an integer? Try running it and see what happens.

In Java, trying to parse an integer from a String that is not an integer throws something called a `NumberFormatException`. In the next section, we will discuss what exceptions are and what you can do about them to make your program secure and functional.

Exceptions

"An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions." - Java Tutorials[1]

Exceptions are a way for your program to communicate an error to the runtime system. It does so by "throwing" an exception object, which holds information about what went wrong. There are

[1] For more information on Exceptions, refer to http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html

many different types of exception objects that are specific to the errors that occur. One example is the `NumberFormatException` mentioned above, which indicates that the program tried to convert a string into a number but that the string does not have the correct numerical format for the conversion. We will look at many other examples of exceptions throughout this lab.

So now that we know what exceptions are, how can we use them to help us with our programs? Exceptions can be quite useful, as they offer specific information about the error that occurs in the program. However, when not handled properly, they'll cause your program to stop. Luckily, Java provides a framework for handling exceptions, and the specificity of the exceptions thrown allows you to customize your response based on the error. Consider another example that, like the age example, requires the user to input an integer. Imagine you are designing a program for a grocery store self-checkout line. You first want to ask the customer to enter how many items they have, to ensure that they don't miss scanning one.

Create a class called `ShoppingCart` and copy the following code:

```
import java.util.Scanner;

public class ShoppingCart {

     public static void main(String[] args) {
          String[] items = getItemsByNumber();
          System.out.println("You have " + items.length +
                             " items in your shopping cart.");
     }

     private static String[] getItemsByNumber() {
          String[] items = null;
          Scanner in = new Scanner(System.in);

          System.out.print("Enter the number of items: ");
          String input = in.next();
          int numItems = Integer.parseInt(input);

          items = new String[numItems];
          return items;
     }
}
```

Try entering the number 8. What happens? Now try entering the word "hi".

You should get the same `NumberFormatException`. Now we will look at how to deal with exceptions in a way that doesn't stop your program.

Try/catch blocks

The main mechanism that Java provides for you to handle exceptions is called a try/catch block. Try/catch blocks are composed of two parts:

- the try block, where you put the code that might throw an exception, and
- the catch block, which checks for the exception and provides information about what the program should do in case the exception is thrown (rather than just quitting as shown).

For example, we might do the following to handle the `NumberFormatException` thrown above.

```
private static String[] getItemsByNumber() {
     String[] items = null;
     Scanner in = new Scanner(System.in);

     System.out.print("Enter the number of items: ");
     String input = in.next();
     try {
          int numItems = Integer.parseInt(input);
          items = new String[numItems];
     } catch (NumberFormatException e) {
          System.out.println("You didn't enter a valid integer.");
     }
     return items;
}
```

At whichever point in the try block the exception is thrown, it jumps to the catch block to be dealt with. In this case, when "hi" is entered instead of an integer, `Integer.parseInt` throws the exception.

In this case, we chose to print a message to alert the user of the error and return normally from the method. Sometimes this is the best option - just to ignore the exception and move on. In this case, run the program and see what happens. Do you see why?

Since the code jumps from the `parseInt` line into the catch block, it skips the line that initializes the items array. Therefore, when `items` returns, it is still `null` from its declaration at the beginning of the method. When we try to print the number of items using `items.length`, we get a very common type of exception. Write the type of exception below:

So how do we deal with this exception? There are a few options. First, now that you know what a try/catch block looks like, add one to the relevant code so that the new exception does not stop the program but instead tells the user "Sorry, you cannot check out." Write the new code below (hint: the code you are changing here is in the `main` method).

But something about this example seems unsatisfying. Do your customers just not get to check out? Wouldn't it be better if they could correct their mistake and enter a valid integer once they realize that they made an error? Luckily, the answer is yes, and it can be done with user input validation.

Input validation

Consider the Shopping Cart example that we have been using so far in this lab. As a refresher, here was the original code:

```
import java.util.Scanner;

public class ShoppingCart {

     public static void main(String[] args) {
          String[] items = getItemsByNumber();
          System.out.println("You have " + items.length +
                             " items in your shopping cart.");
     }

     private static String[] getItemsByNumber() {
          String[] items = null;
          Scanner in = new Scanner(System.in);

          System.out.print("Enter the number of items: ");
          String input = in.next();
          int numItems = Integer.parseInt(input);

          items = new String[numItems];
          return items;
     }
}
```

We tried improving this code to account for the `NumberFormatException`, but what do we do if, instead of just alerting the user of the error, we want to give them the chance to correct it? In fact, we want to keep letting them try until they enter a valid input. This is a process called input validation, and usually involves the following basic steps:

- prompt the user for input
- check that the input is valid - this is either by catching an exception or explicitly checking something about the input
- if the input is valid, continue on to the subsequent code

- if the input is invalid, repeat until it is

This cyclic process is often implemented using a while loop, since we want to continue until we have a valid input. Let's look at how this process would be carried out in the current example.

```
private static String[] getItemsByNumber () {
        String[] items = null;
        Scanner in = new Scanner(System.in);

        boolean inputValid = false;

        while (!inputValid) {
             System.out.print("Enter the number of items: ");
             String input = in.next();

             try {
                  int numItems = Integer.parseInt(input);
                  items = new String[numItems];
                  inputValid = true;
                  //if we made it here, we have valid input
             } catch (NumberFormatException e) {
                   System.out.println("You didn't enter a valid " +
                                      "integer.");
             }
        }

        return items;
}
```

Try replacing the `getItemsByNumber` from the original code with the one written above. Run it and enter an invalid input (such as "hi" again). What happens?

In this case, we keep the try/catch block to check for the exception when we parse the input, but instead of just alerting the user that the input is invalid and moving on, we reprompt them. This allows us to ensure that we get valid input and lets the user make a mistake without stopping the whole program. Notice now that we do not need the try/catch block in the `main` method now, since if the program has returned from `getItemsByNumber` then the array has successfully been initialized.

However we are not totally off the hook just yet. Try running the code but entering "-5" as your integer. What type of exception do you get?

Can you figure out what line of code throws this exception?

It seems we still have more input validation to do. In this case, the fix is relatively simple. We

don't need another try/catch, because we can simply check what the integer is before we initialize the array. Since we already have this loop of reprompting until we set `inputValid` to true, we will include our check in this loop and make sure not to change `inputValid` until the integer is a valid number of items. The while loop now looks like this:

```
while (!inputValid) {
     System.out.print("Enter the number of items: ");
     String input = in.next();

     try {
          int numItems = Integer.parseInt(input);
          if (numItems > 0) {
               items = new String[numItems];
               inputValid = true;
               //if the code made it here, we have valid input
          }
     } catch (NumberFormatException e) {
          System.out.println("You didn't enter a valid integer.");
     }
}
```

Try running the program now. Enter "hi". Now enter "-5". Finally, enter "3" and see your program succeed!

Exercise

To finish this Shopping Cart program, we want to let the user enter their items once they have successfully entered the number of items that they have. Write a method `addItemsToCart` that takes in the `items` array as a parameter and fills it with `String` items entered by the user.

To make things a little more complicated, however, the store has a limited inventory of items, stored in a `Set` that we'll call `options`. You must validate the user input to ensure that the item entered is one carried by the store (i.e. that it is in the `options` set).

Your main method now looks like this:

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

    String[] optionArray = new String[] {"apple", "orange",   "banana",
           "lettuce", "eggs", "milk", "yogurt", "chocolate", "bread", "chicken"};
   Set<String> options = new HashSet<String>();
   options.addAll(Arrays.asList(optionArray));

    String[] items = getItemsByNumber();
    System.out.println("You have " + items.length +
                       "items in your shopping cart.");
   addItemsToCart(items, options);

    System.out.println("Items are: ");
   for (String item : items) {
      System.out.println(item);
   }
}
```

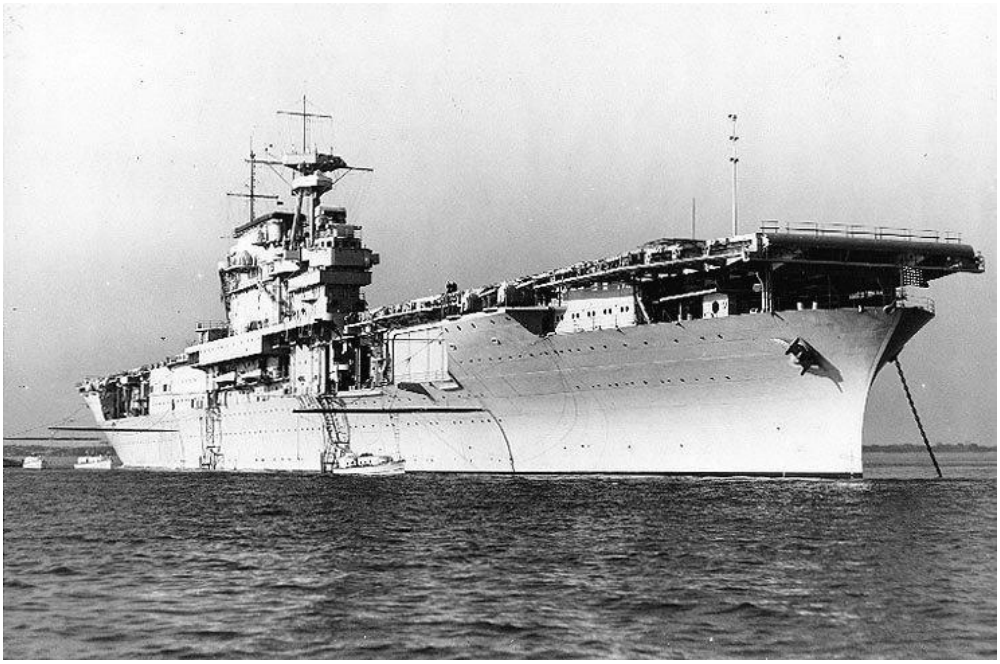
Write the `addItemsToCart` method below. You should prompt the user for input until you have the number of valid items that the user told you they were going to enter (this number can be checked in items.length).

```
public static void addItemsToCart(String[] items, Set<String> options) {

}
```

Exercises

1) **USS Yorktown Software Disaster**



Source: http://ww2db.com/image.php?image_id=6865

In November 1998 the USS Yorktown, a guided-missile cruiser that was the first to be outfitted with "Smart Ship technology", suffered a widespread system failure. "After a crew member mistakenly entered a zero into the data field of an application, the computer system proceeded to divided another quantity by that zero. The operation caused a buffer overflow, in which data leaked from a temporary storage space in the memory, and the error eventually brought down the ship's propulsion system. The result: the *Yorktown* was dead in the water for more than two hours." [6]

Although the details of the exact error that caused this software failure are not public, it is possible that a problem like this could occur from something as simple as an unhandled exception on invalid input.

Consider the following code:

```
import java.util.Scanner;

public class YorktownControl {
	public static void main(String args[]) {
		int speed = 60;
		System.out.print("Cut speed by a factor of: ");

		Scanner in = new Scanner(System.in);
		int factor = in.nextInt();
		speed = speed / factor;
		System.out.println("Your speed is now " + speed);
	}
}
```

Imagine that this is the code that controls the deceleration of the Yorktown ship. If you were to enter 0 as the crewman did, what happens? Record it below.

Now fix the code to validate the input. As we mentioned, in some cases there are 2 ways to approach this: you can either validate the input directly so that no exception will be thrown, or handle the exception using a try/catch. When possible, it is preferred to validate the input directly. Try fixing the code in both ways below:

Input validation:

```
import java.util.Scanner;

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









	}
}
```

Exception handling:

```
import java.util.Scanner;

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









	}
}
```

2) Reading from a file[2]

One time that you will always need to use a try/catch statement in Java is for reading from a file. The following basic code is needed to open a file.

```
import java.util.Scanner;
import java.io.*;

public class ReadFile {
   public static void main(String[] args) throws IOException {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter the name of the file: ");
      String filename = in.next();
      BufferedReader bf = new BufferedReader(new FileReader(filename));
      String line = bf.readLine();
      while (line != null) {
         System.out.println(line);
         line = bf.readLine();
      }
   }
}
```

If the file specified does not exist, it throws a `FileNotFoundException`. Rewrite this code to handle the exception. If the file doesn't exist, keep prompting the user for a new file.

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









}
```

[2] For more information on file reading with BufferedReaders, see http://docs.oracle.com/javase/tutorial/essential/io/buffers.html.

3) Index Problems
This class asks the user for a String, then asks for two indices, and returns the substring between the two indices.

```
import java.util.Scanner;

public class IndexProblems {
	public static void main(String args[]) {
		System.out.print("Enter a string: ");
		Scanner in = new Scanner(System.in);
		String testString = in.next();
		System.out.println("Your test string is: " + testString);

		System.out.print("Enter a start index: ");
		int start = in.nextInt();
		System.out.print("Enter an end index: ");
		int end = in.nextInt();
		String result = testString.substring(start, end);

		System.out.println("Your substring is: " + result);
	}
}
```

What are some possible errors that might occur with this program? If you are stuck, try actually running the programming and testing a variety of inputs.

There are 2 types problems that you want to fix.

First, what happens if the user enters something other than a number for the index?

Second, what happens if the user enters an index that is less than 0?

What about an index that is past the end of the word?

What if the start index is greater than the end index?

You want to validate the users input for two things - the type of input (i.e. an integer) and the values of the inputs (i.e. the range of the integers).

Rewrite this code to handle both of those problems. As a hint, it's easiest to check for the type of

input using a try/catch, similar to what we did in the Shopping Cart example, except this time we are using the nextInt() method directly to get an int instead of parsing the int from the input string. For the values of the input, you can check directly based on the length of the word and the relative values of start and end to prevent an exception from being thrown altogether. Also, note that after reading an invalid input, you will need to call `in.nextLine()` to clear the previous bad input.

```
import java.util.Scanner;

public class IndexProblems {
	public static void main(String args[]) {
		System.out.print("Enter a string: ");
		Scanner in = new Scanner(System.in);









		System.out.println("Your substring is: " + result);
	}
}
```

REFERENCES

[1] “Exceptions Quotes.” Brainy Quotes. Web 5 March 2014. <http://www.brainyquote.com/quotes/keywords/exceptions.html>
[2] “Exceptions.” Java Tutorials. Web 5 March 2014. <http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html>