CS 2300 Week 04 Lab: Java Console I/O using `Scanner` and `println`.

1. Google on: **Java 16 Scanner** (Java 17 Scanner API does not appear to be online yet). The top hit is probably the official Java API (Application Programming Interface) for the Scanner class. There is a lot of info here. Glance through this document to get a feel for what is going on. Because Java 17 is pretty new, you might need to go with Java 16 Scanner. They are probably exactly the same.

There are many ways to get information from your keyboard into a program in Java. Most of these are a lot of work. The Scanner class makes it really simple. Even though we have not studied classes in general, you will see that classes are pretty easy to use!

2. Google on: **Java 17 String**. The top hit is probably the official Java API for the String class. It is very big, but take a brief look at it.

3. Review earlier labs about how to start a project. This writeup assumes you have chosen **NetBeans key mappings**, as discussed in Lab 02. If you do not use them, you can google to find the IDEA equivalents.

4. Start a new JetBrains IDEA project with a name something like **Lab04Scanner**.

Create a class named `lab04.Lab04Scanner` in the project. This class will serve as a **bag** in the sense that it is just a holder for some methods to practice getting input in Java.

5. A class can have one or more **fields**, which can be either a variable of a pdt (primitive data type), or an object of another class. This class is going to contain one field, an object of class `Scanner`. To make this happen, enter the following line of code (just inside the class):

```
private static Scanner scanner = new Scanner(System.in);
```

5.1. Note that you get an error message on the left. Anytime you use a preexisting Java class (such as `Scanner`), you must **import** the class. This means tell the compiler that you want to use stuff from this class, so please "load" the class into the appropriate place. JetBrains IDEA will do this for you. Hit `Alt + Enter` to **fix imports**. Then the `Scanner` class is imported into the class you are writing.

A couple of very basic Java classes, such as `String` and `System`, are automatically imported, so you don't have to import them.

5.2. Because we will only be using one object of class `Scanner`, we use the name `scanner` for the object It is always a good idea to try to find **good names** for things. As far as objects of a class, if you only have one object of a class, give it the same name as the class, but start with a lower case first letter. Less stuff to get confused about is a good thing!

5.3. The word **`private`** means this object can only be used inside this class. Also, this object is **declared** at the top of the class, as opposed to inside a method. Thus it is called a **field**. Sometimes a field is called a "class variable". This means that the entire class can share access to the object. Because it is `private`, code from other classes cannot see anything about this object. This is an important consideration, because controlling how one part of a program can see another part is very important in all programming languages.

5.4. The word **`static`** is there for a subtle reason. Think of it as magic for now.

5.5. The **new** operator tells the **memory manager** to go find some memory where this object can be stored, and store it there.

6. There are many ways to get user input (from the keyboard, mouse, etc.) in Java. The downside of this flexibility is that it is complicated to set any one of them up. For this reason, the Java team language has a class called `Scanner` that makes typical keyboard input easy.

6.1. The code:

```
Scanner(System.in)
```

is one of the many **constructors** (the word "constructor" is usually written as **c-tor** or **ctor**) for the `Scanner` class. Every time you instantiate an object of a class, the c-tor sets up the object so it is ready to go to work. Classes usually have one or more c-tors.

This c-tor takes one argument (input variable) to tell the operating system what to hook the scanner up to. In this case, the argument is: `System.in`, which means: "hook this Scanner up to the keyboard". In many computer languages (including Java), the keyboard is called `System.in`.

All of the stuff going on in this statement

```
private static Scanner scanner = new Scanner(System.in);
```

is referred to as **instantiating** an object (or, sometimes "an instance") named `scanner` of the `Scanner` class that is "hooked up to" the keyboard. Now we can use `scanner` to get input form the user via the keyboard.

6.2. The `Scanner` class can do a lot of work for you here. When you type some number on the keyboard, such as `-123`, `44`, or `1.234E5`, you have actually typed a string of characters that represent a number in "American style", using the "Latin Alphabet", or perhaps using scientific notation (the `E5`). When you ask the object `scanner` to get the next number of some data type, the `Scanner` class knows how to parse (translate) whatever you typed into the primitive data type you ask for. Figuring out how to do this yourself might be a fun project, but we don't have to do it.

7. Below the field `scanner`, there might already be a "main method". If not, create a main method. Either write it out by hand, or use the **shortcut** (type **main**, and double click on the suggestion to create a main method).

7.1. Add some code so that the main method prints out: **Lab 04. Start of Scanner Project**. (Do this exactly as the way you got "Hello, World" to print out). Also add some code near the end of the main method so it prints out **End of Scanner Project. Bye**.

7.2. Hit `shift+F6` to run the program. Be sure it works.

8. **Methods**. Most classes have several methods. A method is a chunk of Java code that does something. One method can call another method to do some task. When you hit `shift+F6`, you "run the main method". Rather than clutter up the main method with a bunch of stuff, we will have the main method call another method to do some work. Do it like this:

9. **Read a line of text**.

Before the main method, write another **method**; like this:

```
public static void demoTextLine(String prompt){
    System.out.println("Enter a line of text. " + prompt + ": ");
    String input = scanner.nextLine();
    System.out.println("Your input text is " + input + ".");
}
```

Details of the `demoTextLine` method:

9.1. The **signature** of the method is `public static void`.

The keyword `public` means any part of the program can use this method.

The keyword `void` means nothing is returned from this method.

Think of the keyword `static` as a magic word that will be discussed later.

The name of the method is `demoTextLine`.

This method has one **argument**, namely a `String` named `prompt`. The role of this argument is to prompt the user (that is you) about what to enter. This is done with

```
System.out.println("Enter a line of text. " + prompt + ": ");
```

9.2. The next statement

```
String input = scanner.nextLine();
```

instantiates a `String` named `input`. Then the method `nextLine` of the `Scanner` class is called. It returns a `String` containing whatever was typed in on the keyboard. Then the **assignment operator =** grabs this stuff, and stores it in the `String` named `input` so it can be used later (or, sooner in this case!).

9.3. The last thing the `demoTextLine` method does,

```
System.out.println("Your input text is " + input + ".");
```

prints out the string `"Your input text is "`, then pastes on whatever is stored in `input`, and finally pastes on a period at the end.

This is the end of the method, so you return to the main method and do whatever is next.

10. **Try it out**. When you are programming, ALWAYS write everything out to make sure it is working!!! You can take these write statements out later to avoid cluttering up the output if you want to.

Go to the main method. Right after the opening print statement, type

```
demoTextLine("one");
demoTextLine("two");
```

Run the program. Read the output area. You will be prompted, like this:

```
Enter a line of text. one:
```

Move the cursor to the end of this line, or just below this line, and enter something, for example `Hello World!`, and for the next one, enter `This is an input string`. Your output should look like this:

```
Start
Enter a line of text. one:
Hello World
Your input text is Hello World!
Enter a line of text. two:
this is an input string
Your input text is this is an input string.
```

This is how you can get data into a running program.

11. **Entering numbers**. Often you need to enter some numbers into a program. If you enter them as strings, then you have to do a bunch of extra work. It is easier to use a special method that reads numbers.

11.1. Put the following method, after the `demoTextLine` method, and before the main method:

```
public static void demoInt(){
    System.out.println("Enter an integer: ");
    int input = scanner.nextInt();
    System.out.println("You entered " + input + ".");
}
```

11.2. This is similar to the `demoTextLine` method, except a method `nextInt` that knows how to **parse** a string into an `int` is used. Note that both methods can have a variable named `input`, because they are in different methods. Technically speaking, they have different **scope**

11.3. Try this out. Go to the main method, and type these lines in:

```
demoInt();
demoInt();
demoInt();
```

Run the program. Additional output will be added on to what is already there,

```
Enter an integer:
444
You entered 444.
Enter an integer:
555
You entered 555.
Enter an integer:
666
You entered 666.
```

12. An **idiosyncrasy**. Here is a keyboard input complication that gets everyone once in a while. Suppose you enter `45` with the keyboard. To do this, you enter `4`, then `5`, and then the `Enter` key at the keyboard. These three characters go into the **keyboard buffer**. When you read the number, the `Scanner` object grabs the `4` and the `5` from the buffer to turn into a number, but it leaves the `Enter` character in the buffer.

This is not a problem if you enter a bunch of numbers. But if you enter numbers, and then read a string, the `Enter` key is still there, and it looks like the end of a string. For this reason, the first string you read in this situation will be empty, which will confuse the programmer! There are other ways to input stuff that avoid this issue.

12.1. Check this out. Go back to the main method, and add

```
demoTextLine("three");
demoTextLine("four");
demoTextLine("five");
```

right **after** the `demoInt` method calls. Now the (end of the)output looks like this:

```
Enter an integer:
666
You entered 666.
Enter a line of text. three:
Your input text is .
Enter a line of text. four:
777
Your input text is 777.
Enter a line of text. five:
888
Your input text is 888.
End. Bye
```

12.2. Observe how the line

```
Enter a line of text. three
```

is followed by

```
Your input text is .
```

This is because the `Enter` char was sitting in the keyboard buffer, and got returned as an empty string, so there is nothing before the period.

12.3. Don't let this issue scare you. This is the nature of working with computers: there are a few things they do that you didn't expect.

13. **Bad input**. What happens if the computer expects a number, but you enter something that is not a number? Let's find out.

13.1. Run the program again. When prompted to enter an integer, enter something that is NOT an integer, like `jkjkjk`. This causes an **exception to be thrown**. This is pretty scary looking:

```
Enter an integer:
kjkjkjk
Exception in thread "main" java.util.InputMismatchException
     at java.base/java.util.Scanner.throwFor(Scanner.java:939)
     at java.base/java.util.Scanner.next(Scanner.java:1594)
     at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
     at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
     at lab04.Lab04Scanner.demoInt(Lab04Scanner.java:18)
```

```
    at lab04.Lab04Scanner.main(Lab04Scanner.java:30)

Process finished with exit code 1
```

13.2. Don't let this scare you. Notice the very top line says: "java.util.InputMismatchException". The key piece of information is **Mismatch**. That is a clue that a bad number was typed in.

13.3. **Throwing exceptions** is one way of dealing with bad input. We will discuss them later. For now, look at the output the exception machinery generates. At the top is the type of exception. The name of the exception should give you a clue as to what is wrong: **java.util.InputMismatchException**. "Input Mismatch" should give you a good idea what went wrong.

Next comes a list of line numbers in classes, some of which might be built-in Java classes (part of the Java language). The first line number listed is where the exceptional condition was detected. The last line number listed is the line in your code that is triggering the problem. This list helps you figure out what went wrong so you can fix it. In this case what went wrong is someone entered bad input, so you don't have to fix the code, just try running it again.

13.4. Notice the last line:

```
Process finished with exit code 1
```

Usually we see

```
Process finished with exit code 0
```

What is the deal with this? This is kind of a historical artifact going back more than 50 years. When programs are done, they give out an integer that is called the **exit code**. If everything worked, the exit code is `0`. Otherwise it is something else (not `0`). Long ago, there were tables of number that indicated different things that went wrong. That system did not work very well, so now usually the error code is `1` for anything that goes wrong.

13.4. In C++ programming, the names `EXIT_SUCCESS` and `EXIT_FAILURE` are used rather than `0` and `1`.

14. Now add another method called `demoDouble`. This should be about the same as `demoInt`, but make whatever changes are needed to work with a `double` instead of an `int`..

14.1. Try this method out with: `123.45`, `123`, `123.`, and `123.45E5` for input, and see what happens. This gives an idea of how it works.

15. **Submit your lab**. Submit the same way as the other labs. These will be graded online.