CS 2300

Project 1: Methods – Output – Strings – Operators

Due: Next Wednesday, October 5, by 11:59 PM

This project is intended for practice on basic Java tasks that have been introduced in class. Each task has a small chunk of code to implement it, and some follow up discussion. Tasks include the following:

Gain experience in completing a project. Hint: start early.

Make you programs work. Hint: start small, add one thing at a time.

Break the Java code into small methods, to simplify the development of the program.

Start small, add one thing at a time. Compile. Fix if needed. Test. Fix if needed.

Use math operators.

Format output.

Creating programs by incremental development:

** **Start small, add one thing at a time. Ensure that it works before continuing**. **

In other words, make sure each chunk of code that you add works correctly before going on to the next addition. Don't forget this!

If you don't program this way, you will get a big mess with lots of errors. No one will be able to help you get it to run. You will have to pitch it and start over. Next time start small!

Small methods make the work easy:

The easy way to write programs is to break the code up into small methods. A large program should be one main method, calling a bunch of small methods, each of which might call other small methods.

Break everything down into small easy methods that you can understand and test.

After each addition, run the program and be sure it works. Fix any errors. Run it again. Fix any errors. Repeat as needed. Go on to next step. Fixing errors is called **debugging**. Debugging is much easier if you do it continuously as you write the code. JetBrains IDEA compiles on the fly and tells you about any syntax errors. You still need to run the program every time you add anything so you can be sure it works correctly. A common rookie mistake is to write a whole program and then try to debug it afterwards. This is way too hard; you will never get it fixed, so you have to start over. **Don't do this!** (And if you do, don't bring it to me for help!)

1. Start:

1.1. Fire up JetBrains IDEA.

Start a new project with a good name, such as `Project1`.

1.2. Create a class and package:

A good package/class name combo would be `proj1.Project1`. (Don't put the final . in the name, that is a period)

In the edit window an empty class should appear:

```
public class Project1 {

}
```

1.3. Put in a main method.

1.4. Add the lines

```
System.out.println("Project 1");
System.out.println("Your name here");
```

inside the method. (Make sure **your** name is in the print statement).

Run the program to be sure it is working. Enter shift + F6 to run the program. An output window should open below the editor with the output `Project 1` on it, along with some system info.

1.5. If anything went wrong, figure out how to fix it.

If things went REALLY wrong, and you cannot fix it, start over. Before you can start over, you should clean up the mess from the first try by having JetBrains IDEA delete everything. (Or, if you are lazy, you can just pick another project and package name). Then start over.

The best way to clean up a dead project is to right click on the project name (in the projects window), then click on delete, to totally clean up the project. It is a big mistake to try to delete a project yourself by working with the computer's file system, there are many files in every project, and you will likely miss some. Let JetBrains IDEA do the work (this is an example of **refactoring**, a JetBrains IDEA assisted major change in your project).

1.6. : Put in a **JavaDoc comment** above the class name, something like this:

```
/**
 *
 * @author (c) Ralph Kelsey, Ohio Univ Comp Sci Dept, 2022
 */
```

Be sure your name is in it (do not put Ralph Kelsey in it. I think that I am the only Ralph Kelsey in this class!).

2. String stuff. In Java there are two types of strings:

C – strings are sequences of characters surrounded by a pair of parentheses, i.e.: `"this is a c string"`. We have been using these since day 1.

String class strings. Objects of the `String` class are smart strings and can do lots of cool things.

For a warm up, Google on `Java 17 String`, and check out the API. Don't worry about understanding everything, just start to get familiar.

2.0. Add some demonstrations to the project. Each of these will live in one or more methods. These will be **static methods**, which mean they do not depend on any "class fields". This project is implemented as a "bag of static methods", which is how projects in C are usually written. We will keep the main method last.

The first demonstration method is for the built in `String` class.

2.1. Above the main method, type in

```
private static void stringStuff(){
    System.out.println("\nString Stuff\n");
}
```

This is a **header** for this method. We will add stuff to it presently. The **access specifier** is `private` (rather than `public`) for this method because only other methods in this class will call the `stringStuff` method. Making this `private` instead of `public` is part of the goal of making everything as protected as possible.

2.2. Add a line to the `main` method, so it looks like this:

```
System.out.println("\nProject 1.\n");

stringStuff();

System.out.println("\nBye.\n");
```

Here the `main` method is **calling** the `stringStuff` method.

2.3. Enter shift + F6 to run the program. When you enter shift + F6, the main method in the current class runs. (The "current class" is the one showing in the editor. It is the only class in this project, but often there will be a bunch of classes)

The statement `stringStuff();` calls the `stringStuff` method. This means go to the `stringStuff` method, and do whatever is in it. All we are doing at this point is ensuring that both methods ran. Check out the output to see that this is the case.

2.4. Now add the following lines to the `stringStuff` method. BE SURE to replace Kelsey and Ralph with YOUR OWN NAME. You can use multiple word names, and put it in the order you want.

```
String lastName = new String("Kelsey");
String firstName = "Ralph";
String fullName = firstName + " " + lastName;
System.out.println("my name is: " + fullName);
System.out.println("I said:     " + fullName.toUpperCase() + "!");
```

Run this and check it out. Guess what the `toUpperCase` method does? Be sure you understand what is going on in each line.

2.5. Above are three **objects** of the **String** class. Each of these objects has the word `String` in front. This is an example of **strong typing**, which means that each variable (either a pdt or a class) must state what it is so that the compiler can check out that it is not used incorrectly. This is an enormous help, because it keeps programmers (you, me, everyone) from making all kinds of mistakes.

The three `String` instances are each **instantiated** a different way for illustrative purposes.

2.5.1. The name of the first object is `lastName`. This object is **instantiated** with a usual Java **constructor**, which is what the second occurrence of the word `String` is. The word `new` tells the **memory manager** to find a place to store this object.

2.5.2. The name of the second object is `firstName`. This illustrates a different way of instantiating a `String` object. Because so many objects of type `String` are created, the compiler will put the `new String` code in automatically for us, as shown.

2.5.3. The name of the third object is `fullName`. We do not have to instantiate this object, because that work is done by the `String` class when it sees the + operator. In this situation the + operator is used for **concatenation** (joining two strings together). The code

```
firstName + " " + lastName
```

is actually a shortcut for:

```
firstName.concat(" ").concat(lastName).
```

This combination produces a new object of the `String` class, which we store in the `fullName` object. A lot of work is being done pretty much effortlessly.

2.6. **The toString method**. In Java, every variable or object has a `toString` method. This method knows how to turn the variable or object into a string whenever needed. This is a totally cool system! Here is an example:

```
int num = 2;
System.out.println("Hello");                           // print line 1
System.out.println(num);                               // print line 2
```

The `println` method gets one **argument** sent to it. This argument must be a string of some kind. Here is what is happening above:

Print statement 1: The argument is the string `"Hello"` . The `println` method then prints this argument on the monitor.

Print statement 2: The compiler expects the argument to the `println` method to be a string, but it is an `int`. Whenever a string is expected, but something else is there, the compiler finds the appropriate `toString` method to turn the argument into a string. You can think of this as "everything knows how to turn itself into a string". The windup is, you see the value of `num`, namely `2`, get printed out.

2.7. **Overloaded + operator**. Overloaded means that the same symbol or method means different things in different contexts. We have seen this with arithmetic, where 5 / 2 has a different meaning than 5.0 / 2. The correct meaning is determined by the **operands**, i.e., the things to the left and to the right of the operator.

In addition to arithmetic, there is one more overloaded operator in Java – the + operator can mean addition of numbers, or concatenation of strings.

Which + operator is it?. Look to the left of the + operator. If the thing to the left is a number, the + means addition. If the thing to the left is a string, the + means concatenation. This is illustrated by the next example.

Add two more lines to the above example to get:

```
int num = 2;
System.out.println("Hello");                          // print line 1
System.out.println(num);                              // print line 2
System.out.println("22: " + num + num);               // print line 3
System.out.println("4:  " + (num + num));             // print line 4
```

Here is what is going on with the additional statements.

Print statement 3: There are two + operators. Look at the first one (going left to right). To the left of the + operator is a string, so the + means concatenation. So the variable `num` is translated into the string "2", and is pasted onto the end of the string `"22:  "` to obtain the string `"22:  2"`. Next look at the right + operator. To the left is a string `"22:  2"`, so again `num` is translated into the string "2" and pasted onto the end of the other string to get the string `"22:  22"`., which is then printed out.

Print statement 4: This one is different because there are a pair of **parentheses ()** in it. Stuff inside parentheses always gets done first. So `(num + num)` gets done first. In this case, the + operator looks left and sees an `int`, so it does arithmetic, and the expression `(num + num)` gets translated into an `int` with the value `4`, resulting in the expression `"4:    " + 4`. Next the other + operator looks left and sees the string `"4:    "`, so the + is concatenation. Thus the appropriate `toString` method turns the number `4` into the string `"4"`, and is pasted onto the end of `"4:    "`, resulting in the string `"4:    4"`. This is what gets printed out.

Hope that wasn't too confusing. You will find that this system is really easy to use, and usually everything works the way you want it to.

2.7. **Popup menus**. Here is how to see everything that you can do with a `String`. The same can be done with every other Java class.

After the above four print statements, start another line with

```
fullName
```

Now add a dot at the end, to get

```
fullName.
```

As soon as you add the dot, a menu with about 50 methods for the `String` class appears. Read through it to develop your intuition for using the Java `String` class for working with strings. Manipulating strings is an important part of programming. It is **much** easier in Java than in most other computer languages. This line is just to demonstrate what happens when you type in a dot, so erase it by entering ctrl + E.

Now erase `fullName.`, that was just for showing what the dot does.

3. Number Output.

3.1. Similar to the last part, add another method called `numberOutputStuff`. This method should also be `private`. Start with a line printing out the title.

In the main method , add a statement

```
numberOutputStuff();
```

to call the new method. Try it out to be sure it runs.

3.2. Now add some code to the new method:

```
double oneThird = 1.0 / 3.0;
System.out.println("oneThird = " + oneThird);
```

3.2.1 Run this and check it out. The number 1/3 cannot be written exactly as a decimal. It also cannot be represented exactly as a `double`. The Java standard output gives about 16 significant digits. This is about the full accuracy of the internal representation of a `double`.

3.3. But you might not want to see all of these decimal places for 1/3. Java provides extensive support for writing numbers (and other things) in whatever way you want. These tend to be rather involved. A particularly easy way to write numbers in any desired format is to use old fashioned C style `printf` (print formatted) output. This was added to Java about 15 years ago.

3.3.1. Add the following code after the last code you entered:

```
        System.out.println("");
        System.out.printf("%f", oneThird);
        System.out.println("");
```

3.4. Here we are using the Java `printf` method instead of the Java `println` or `print` methods. The `printf` method is a slightly improved version of the old C `printf` function.

3.4.1. How `printf` works. Recall that **arguments** are pieces of information sent to a method. This method has **one or more** arguments. It always has a **format string** argument, plus zero or more additional arguments.

The format string argument comes first, and it is a **c string**, i.e., something surrounded by a pair of double quotes, such as: `"I am a c string"`.

The string could be something simple like `"Hello World!"`, which will print the format string as is. If it is `"\nHello World!\n"`, it will start a new line, then write `Hello World!`, and then start another new line.

Often the format string contains one or more **format rules**, which control how numbers and other things are printed. These are in addition to possible text to be printed as is.

The example above uses the format rule `"%f"`. A format rule starts with a `"%"` and is sort of a magic placeholder to print the value of a variable in. For example:

`"%d"` means “print one integer in decimal (base 10) form”, and

`"%f"` means print one floating point number here.

The space allocated by a formatting rule is often called a **placeholder**. Some people call these a **mask** or a **field**. They will be called placeholders in the discussion below.

3.4.2. In the example above, the `double` stored in the variable `oneThird` is printed with the rule `"%f"`, which means write it out as a standard FP number using the default number of decimal places.

Also, there is NO automatic new line. Anywhere you want a new line to start, put in a `"\n"`. The character `\n` is the ASCII symbol for “new line”.

Here is another example, with three variables (two `int`s, one `double`) being printed:

```
int a = 2;
int b = 3;
double x = 0.66666;
System.out.printf("\nFraction: %d / %d = %f", a, b, x);
```

This should print:

```
Fraction: 2 / 3 = 0.66666
```

starting on a new line. Here is how the format string made this happen:

`\n`	New line character (so output starts on a new line).
`Fraction:`	Write the text `Fraction:` and skip a space.
`%d`	This rule is the first of three rules, so the first argument, `a`, will be written as a decimal integer.
`/`	Skip one space, print a `/`, skip another space.
`%d`	This is the second rule is, so the second argument, `b`, will be written, again as a base 10 integer.
`=`	Skip one space, print an `=`, skip another space.
`%f`	This is the third rule is, so the third argument, `x`, will be written as a floating point number.

This system is very flexible and not hard to figure out. Be sure you understand how this works.

3.4.3. So far, `printf` does not look like much of a big deal, but you can also put spacing information in the place holders. Some important examples are:

`%6d`	print an `int` in 6 places, **right justified** pushed to the right.
`%-6d`	print an `int` in 6 places, **left justified** (pushed to the left).
`%12f`	print a `double` in 12 places, right justified (which is the default).
`%12.4f`	print a `double` in 12 places, with 4 digits after the decimal point, right justified.
`%12s`	print a `string` in 12 places, right justified.

This easy number formatting is very useful when making tables of numbers, etc.

3.5. To illustrate how a number gets written in a placeholder, we will print out some numbers, each surrounded by a vertical line (BTW, the vertical line is called "pipe"). The pipe symbols are so you can see how the spacing works.

Add the line

```
System.out.printf("\n|%f|", oneThird);
```

This means: (1) start on a new line with `\n`, (2) write a `|` to locate the start of the placeholder, (3) write the floating point number in `oneThird`, (4) draw another `|` to locate the end of the placeholder. Check this out. Now there are only 6 decimal places written out.

The reason there are only 6 significant digits is because the `printf` statement is adopted from C and C++, where 6 is the default. That way `printf` works the same in all computer languages.

It is different with `System.out.println`, which uses 15 significant digits, which is the default in Java.

3.6. Next copy and paste the last line several times, and adjust to create the following:

```
System.out.printf("\n|%20.2f|",  oneThird);
System.out.printf("\n|%20.12f|", oneThird);
System.out.printf("\n|%20.20f|", oneThird);
System.out.printf("\n|%-20.2f|", oneThird);
System.out.printf("\n|%20.0f|",  oneThird);
System.out.printf("\n");
```

This pretty much shows everything that can happen. As soon as you understand what is going on here, you are an expert on `printf`!

Now add these two slightly different lines (careful to get them exactly right):

```
System.out.printf( "\nC/C++ default: |%f|", oneThird);
System.out.println("\nJava default : |" + oneThird + "|");
```

These last two lines illustrate the difference between the default in C and C++, and the default in Java.

4. Integer Math Operator Table. This will be a simple table showing how the basic math operations work for integers. This is a good example of how you should learn any new programming feature; print out a table of what it does. Then you can read it and see if it agrees with what you think it should be. When the computer disagrees with you on what something should look like, either you are wrong, or the computer is wrong! In my experience, 100% of the time it is me that was wrong (HaHa!).

4.1. As in Section 3, add another method called `intMathTable`. We will start with a **stub version**. That means a mini version of the method to get started. Start with one line in this method, printing out the title: `Integer Math Table`.

4.2. Shortcut alert! Now that you have typed out `System.out.println()` enough times, you can be let in on the secret that there is a shortcut for it (and many other things). Click Help/Keyboard Shortcuts Card. You can print this out if you want. Note it is two pages long. The end of page 1 and most of page 2 are “Java Editor Code Templates”. Note that one of them is **sout**. Try typing `sout` and then hit the tab key. This is probably the best shortcut in the entire bunch.

4.2. The full `intMathTable` method will use two **helper methods**; `mathLine` and `headerLine`. Put the code for these two methods **before** the code for the `intMathTable` method (because the two new methods are used by the `intMathTable` method). You don’t have to do this, but it is a good idea.

Here is the first method:

```
private static void headerLine(){
    System.out.printf("\n");
    System.out.printf("%10s", "a");
    System.out.printf("%10s", "b");
    System.out.printf("%10s", "a * b");
    System.out.printf("%10s", "a / b");
    System.out.printf("%10s", "a % b");
}
```

Notice the only `"\n"` is in the first line, so this will print a one line header for some columns to be printed subsequently. The header just indicates what math operations are being done.
Here is the second method:

```
private static void mathLine(int a, int b){
    System.out.printf("\n");                    // new line
    System.out.printf("%10d", a);               // write a
    System.out.printf("%10d", b);               // write b
    System.out.printf("%10d", a * b);           // write a * b
    System.out.printf("%10d", a / b);           // write a / b
    System.out.printf("%10d", a % b);           // write a % b
}
```

The `mathLine` method takes two `int` arguments; `a` and `b`. Each time it is called, it starts a new line, and writes out five numbers, each in a field of width 10. The `headerLine` method is there to print column headers. Notice how the two methods are written **exactly** the same, to ensure that the output lines up correctly.

This way of writing methods makes it easy to make adjustments (such as column width) as needed. For example, if we tried this out and the output looks kind of cramped, we can chance all of the `10` 's to `12` 's to give all the numbers some space.

4.3. Now upgrade the `intMathTable` like this:

```
private static void intMathTable(){
    System.out.println("Integer Math Table");
    System.out.println("");
    headerLine();
    System.out.println("");
    for (int i = 0; i < 12; ++i){
        mathLine(i, 5);
    }
    System.out.println("");
}
```

First the `headerLine` method is called to write the column headers. Then the `mathLine` method is called inside a loop to create as many lines as we want. A couple of blank lines are also written to make the table look professional.

4.4. Add the following feature. Change the two helper functions so that there are six columns, with a new column for **a + b** before the **a * b** column. Be sure the header correctly matches the numbers.

4.5. Change the width of the fields (numbers and column headers) from 10 to 12.

4.6. In the loop in function `intMathTable`, make the values of the `int i` go from `0` to `20`. This takes one slight change in the code.

4.7. Put some extra `System.out.println("");` statements in the main method so that the several demonstrations are separated by a couple of lines (as is, they might be all jammed together).

4.8. Run the program. If the output does not look good, fix it.

What to submit.

The Java class you wrote. Be sure your name is in the comments at the top of the code.

Submit this file to the Blackboard assignment option (same as with labs), as follows.

Be sure you submit files with a name like `myFile.java`, and **not** `myFile.class`, which is the compiled version and cannot be read by human beings.

These will be graded online.

Grading.

1. 25 points consisting of about 5 points for your code, and 20 points for the output your code produces. The 5 points for code is subject to being **dinged** for bad formatting, or forgetting to put your name at the top, and stuff like that.

2. Late projects will be dinged two points after 12:00, and an additional point about every six hours. Thus your score will be all the way down to 0 in about four days. “Dinging” is the traditional way to teach students to submit stuff on time!