LAB 04 TABLE OF CONTENTS

4.1 The Primitive Types
4.2 Data Type Conversions
4.3 Arithmetic Operations and Promotion
4.4 Writing a Complete Program
4.5 Investigate DecimalFormat
4.6 Math Class Methods

PURPOSE OF LAB 04

The Java language defines eight basic types that are called the primitive types. These types are not defined by classes and, therefore, variables of these types are not objects. These types are the necessary building blocks for all programs and represent numbers, both integers and floating-point types, single characters and the Boolean values true and false. In this lab you will investigate the numeric primitive types, data type conversions from one primitive type to another, the division and modulus operations and formatting numerical output.

TO PREPARE FOR LAB 04

- Read Wu: Chapter 3 pages 86 – 116
- Read through this laboratory session
- Using your memory device, create a directory called a4 and copy the five files MinMax.java, Convert.java, Arithmetic.java, Coffee.java and FormatTest.java from http://www.mscs.mu.edu/~marian/Labs/lab04/. All work for this lab should be saved in this directory.

TO COMPLETE LAB 04

- This is an individual lab. You may ask the lab tutor for help and you may consult with your neighbor if you are having difficulties.
- In this lab, you will modify four of the copied programs that are stored on your memory device: do not store these programs on the PC. When done with each program, copy the code and paste it into a file called Lab04.txt followed by the output of the final run of the program.

When you have finished this lab, hand in a printed copy of your Lab04.txt document.
4.1 THE PRIMITIVE TYPES

Program: Open the program MinMax.java. Read the code for understanding.

The primitive integer types: long, byte, short, int

In Java, the four primitive integer types are long, int, short and byte, which are stored in 8, 4, 2 and 1 bytes, respectively. Each of these types can store a largest and smallest value. For example, a variable of type byte is stored in eight bits, which can store \(2^8 = 256\) bit patterns. Half of these bit patterns, 128, are used to represent negative integers (-1 to -128) and the other half are used to represent positive integers and zero (0 to 127). Using Two's Complement notation, 0 is stored as 00000000, 127 is stored as 01111111, -128 is stored as 10000000 and -1 is stored as 11111111.

Each of these primitive types has a related class type, Long, Integer, Short and Byte defined in the Java API package java.lang. These classes are called wrapper classes because they "wrap" a primitive type with some useful methods. They also define class constants called MAX_VALUE and MIN_VALUE, which store the maximum and minimum values for the corresponding primitive type. To access a class constant dot the constant name with the name of the class.

ClassName.CONSTANT_NAME

Step 1: Read the code in MinMax.java that prints to the console, System.out. The java.lang package is automatically imported into every Java program because it contains basic Java classes such as the String and wrapper classes.

```java
//   Name:
//Teacher:
//Section:
//Program: MinMax.java prints the minimum and maximum values
//         that can be stored in types byte, short, int and long
class MinMax
{
    public static void main(String[] args)
    {
        System.out.println("\nMinimum byte value: " + Byte.MIN_VALUE + "\nMaximum byte value: " + Byte.MAX_VALUE);
    }
}
```

Compile the code and run the program. Record the results in the table which lists the integer types in order from highest to lowest precision.

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
<th>MIN_VALUE</th>
<th>MAX_VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 2: Complete the table in Step 1 by adding code to MinMax.java to find the minimum and maximum values of the other three integer types. Compile and run the program. Record the results in the table.
The primitive floating-point types: double, float

A computer also stores real numbers, or floating-point numbers. Java provides two primitive data types to store real numbers, double and float, which are stored in 8 and 4 bytes, respectively. And, in the java.lang package the Double and Float classes define the constants MIN_VALUE and MAX_VALUE.

Step 3: Add code to MinMax.java that prints the minimum and maximum values that can be stored in types double and float. Record the results in the table below.

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
<th>MIN_VALUE</th>
<th>MAX_VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notice, that MIN_VALUE is a minimum magnitude, or closeness to zero; both of these types can store negative numbers. Type double can store larger numbers than float and also numbers that are more precise. A floating-point number stored in eight bytes has double the precision of one stored in four bytes, hence the name double. Unless the amount of available memory is a concern, floating-point variables should be declared to be of type double.

The other two primitive types: char, boolean

The remaining two primitive types are char, which is used to store characters according to the Unicode character set, and boolean, which is used to store one of two values: true or false. We will experiment with char in the next program. For now, try to do the following.

Declare two boolean variables: isFun and isBoring. Assign to one of the variables the value true and to the other variable assign false. Then, write a print statement that uses these two variables in a sentence. Possible output could be ☺

I find it to be true that Java is fun and false that Java is boring.

(Optional) An interesting loop

In Lab 02 we used machine language to write a looping structure and also saw how we could write such a structure in Java. In Wu, loops are discussed in Chapter 6 but, we can look at a short one now. The loop

```java
int count = 0;
int n = Integer.MAX_VALUE - 1;
System.out.println();   // prints a blank line
while(count < 4)
{
    System.out.print(n + " "); // print does not print a newline and
    n = n + 1;                  // leaves the cursor on the same line
    count = count + 1;
}
System.out.println();   // prints the newline
```

Initially count is 0 and n is one less than the maximum value that can be stored in an int variable.

- Since count < 4 is true, n is printed, n is increased by one, and count becomes 1.
- Since count < 4 is true, n is printed, n is increased by one, and count becomes 2.
- Since count < 4 is true, n is printed, n is increased by one, and count becomes 3.
- Since count < 4 is true, n is printed, n is increased by one, and count becomes 4.
- Since count < 4 is false, the loop ends.
**Step 4:** Add the above code to the end of main. Compile and run the program. Record and explain the results.

---

Add MinMax.java and one run of the final program to Lab04.txt

### 4.2 DATA TYPE CONVERSIONS

**Program:** Open the file Convert.java. Read the file for understanding.

From highest to lowest precision (from the widest to the narrowest) the six numeric types are:

```
    double float long int short byte
```

The floating point types have higher precision than the integer types since every integer can be stored as a real number (45 can be stored as 45.0), but a floating-point number can’t be stored as an integer without a loss of information, or precision (at best 2.25 can be stored as 2).

Integer literals, such as 128, are always created as type int. Therefore, the statement

```
    int anInt = 128;
```

stores these four bytes 00000000 00000000 00000000 1000000 in the memory location anInt.

Frequently, one type of data must be converted to another type of data. An **implicit conversion** is a conversion that is done automatically. An **explicit conversion** is one that must be forced by the programmer. What happens when the four byte 128 is assigned to a variable of one of the other integer or real types? We will investigate this and more in the program Convert.java.

**Step 1:** The code declares and initializes six variables, one of each of the six primitive, numeric types. The print statement prints each of the six values.

```java
class Casting
{
    public static void main(String[] args)
    {
        int anInt = 28;
        long aLong = anInt;
        short aShort = anInt;
        byte aByte = anInt;
        float aFloat = anInt;
        double aDouble = anInt;

        System.out.println(
            "\nLong   = " + aLong + "\nShort  = " + aShort + "\nFloat  = " + aFloat + "\nDouble = " + aDouble); 
    }
}
```
Compile the code and record the "essence" of the two error messages.

_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________

A **widening conversion** is an automatic conversion that occurs when a value of lower precision is assigned to a wider, or higher precision, variable.

A **narrowing conversion** must be forced by the programmer when a value of higher precision is assigned to a narrower, or lower precision, variable. In Java, a narrowing conversion must be explicitly forced by the programmer with a **type cast**.

(data type) expression

**Step 2:** To correct the two errors in the above code, explicitly cast anInt to a value of type short and a value of type byte. Revise the statements that did not compile to include the type cast

```java
short aShort = (short) anInt;
byte aByte = (byte) anInt;
```

Compile and run the program. Record the results.

_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________

Casting the int as a short copies the two right most bytes, i.e. the two least significant bytes

```
00000000 00000000 00000000 00011100
```

into the two bytes of memory reserved for aShort. Similarly, the least significant byte is copied to the one byte reserved for aByte.

**Step 3:** Change the value that is assigned to anInt from 28 to 128. Compile and run the revised program. Record the results.

_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________

Using what you know about the storage of integers using Two's Complement notation, explain the strange result in the loss of precision when 128 is cast as a byte.

_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
Step 4: In Java, floating point literals such as 8.75 are created as type double in eight bytes. Add this code to the end of main in Convert.java.

```java
adouble = 8.75;
afloat = 8.75;
System.out.println("\nadouble = " + adouble + "  afloat = " + afloat);
```

Compile the file and record the "essence" of the error message.

_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________

Floating-point literals will be created in four bytes if they are followed by the letter f or F.

Step 5: Revise the statement

```java
afloat = 8.75;
```

by adding an F

```java
afloat = 8.75F;
```

Compile and run the corrected program. Record only the results of the new code.

_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________

Step 7: If instead the right side of the assignment statement is a variable of type double, an explicit cast must be used. Add the following code to the end of the main method.

```java
adouble = 123456789.0123456789;
afloat = (float) adouble;
System.out.println("\nadouble = " + adouble + "  afloat = " + afloat);
```

Compile the updated file. Run the program. Record the new output.

_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________

The number 123456789.0123456789 was chosen so that you can easily count the number of digits that remain accurate, that is the number of significant digits, when the variables are stored and printed. How many significant digits are printed for adouble and afloat?
Step 8: What happens if a real value is cast as an integer? Add the following to the end of the main method.

```java
aDouble = 8.75;
```

Now, add four statements that assign `aDouble` to each of the four variables `aLong`, `anInt`, `aShort` and `aByte`. Follow these statements with this print statement.

```java
System.out.println("\n\naLong   = " + aLong + "    anInt   = " + anInt + 
"\n\naShort  = " + aShort + "    aByte   = " + aByte);
```

Compile the expanded code. Run the program and record the new results.

When casting a real value to an integer value, are the values rounded to the nearest integer or truncated by removing the fractional portion?

Step 9: In Java, characters are stored in two bytes as integer values, according to the Unicode encoding scheme. To see how the character 'B' is stored, the character can be cast as an `int` for printing. Add this code to the end of the main method.

```java
char aChar = 'B';
n1 = 48;
System.out.println(aChar + " is stored as the integer " + (int)aChar);
System.out.println(n1 + " represents the character " + (char)n1);
```

Compile the expanded code. Run the program and record the new results.

Add Convert.java and one run of the final program to Lab04.txt
4.3 ARITHMETIC OPERATIONS, PRECEDENCE AND PROMOTION

Program: Open the file Arithmetic.java.

Implicit conversions also occur when doing arithmetic. The binary operators, (+, −, *, /, %,) work with two operands. If the operands are of the same data type, double, float, long or int the resulting value is of the same data type. Thus, the addition 3 + 5 is performed as the sum of two int values and has int value 8. The addition 3.0 + 5.0 is performed as the sum of two double values and has double value 8.0. When the data types are of mixed types, the lower precision value is promoted to a value of higher precision type when it is loaded in the CPU. Thus, 3 + 5.0 is evaluated as 3.0 + 5.0 and has value 8.0.

There are two types of division, integer and floating – point division.

- Integer division occurs when both operands are integers.
- Floating point division occurs when at least one of the operands is a floating – point value.

Step 1: Read the code in the file Arithmetic.java for understanding.

```java
class Arithmetic {
    public static void main(String[] args) {
        int n1, n2;
        double d1, d2, d3;
        System.out.println("\n4/5   = " + (4 / 5) + " 16/5   = " + (16 / 5) + " \n4/5.0 = " + (4 / 5.0) + " 16.0/5 = " + (16.0/5));
        n1 = 17 / 3;
        d1 = 17 / 3;
        n2 = 17 / 3.0;
        d2 = 17 / 3.0;
        System.out.println(n1 + "\t" + d1 + "\n" + n2 + "\t" + d2);
    }
}
```

Compile the code and record "essence" of the error message.

_____________________________________________________________________________________________
_____________________________________________________________________________________________

Step 2: Correct the line that contains the error by casting the results of the calculation, as an int. Predict the output of the program by filling in the blanks.

Predict the output of the program by filling in the blanks.

<table>
<thead>
<tr>
<th>4/5</th>
<th>16/5</th>
</tr>
</thead>
<tbody>
<tr>
<td>4/5.0</td>
<td>16.0/5</td>
</tr>
</tbody>
</table>

Step 3: Compile and run the corrected program. Check your predicted results, noting the correct answer where needed.
**Step 4:** Add this code to the end of the main method.

```java
n1 = 12;
n2 = 5;
d1 = n1 / n2;
d2 = (double) (n1 / n2);
d3 = (double) n1 / n2;
System.out.println("Step 4: " + d1 + "  " + d2 + "  " + d3);
```

Predict the output of the new print statement.

---

Compile and run the modified program. Check your predicted results, noting the correct answer where needed. Noting that casting has higher precedence than the arithmetic operations, explain the three results.

- `d1 = n1 / n2`
- `d2 = (double)(n1 / n2)`
- `d3 = (double) n1 / n2`

**Step 5:** Modify the values of `n1` and `n2` several times until you are sure that you understand the difference in the three calculations above. One such test could be `n1 = 12` and `n2 = 100`. Record your results and comments.

---

**Step 6:** To be sure you understand the result of the modulus operation, enter the following at the end of main.

```java
n1 = 12;
n2 = 4;
System.out.println(
    n1 + "/" + n2 + " = " + (n1 / n2) + " \t" +
    n1 + "%" + n2 + " = " + (n1 % n2));
```

Predict the output of the last print statement.

---

Compile and run the modified program. Check your predicted results, noting the correct answer where needed.
**Step 7:** Either modify the values of n1 and n2 in the statements you entered as part of Step 6 and run the program multiple times. Or, copy and paste the code multiple times, change the n1 and n2 values and run the program once. Complete the chart:

<table>
<thead>
<tr>
<th>n1</th>
<th>n2</th>
<th>n1 / n2</th>
<th>n1 % n2</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Step 8:** In algebra, $3 \times 5 / 2 \ (15 / 2 = 7.5)$ is equivalent to $5 / 2 \times 3 \ (2.5 \times 3 = 7.5)$. Add the following code to the end of main.

```java
d1 = 3 * 5 / 2;
d2 = 5 / 2 * 3;
System.out.println("\nStep 8: "+d1+" " +d2);
```

Predict the output of this code first.

Compile the code and run the program. Correct your predictions if needed. Explain the results.

**Step 9:** Add the following code to the end of main.

```java
d1 = 3.0 * 5 / 2;
d2 = 5 / 2 * 3.0;
System.out.println("\nStep 9: "+d1+" " +d2);
```

Predict the output of this code first.

Compile the code and run the program. Correct your predictions if needed. Explain the results.

---

Add Arithmetic.java and one run of the final program to Lab04.txt
4.4 WRITING A COMPLETE PROGRAM

Problem: A coffee company wants a program that receives an order for bulk coffee from a customer and prints out an invoice. Coffee sells for $8.99 a pound and a sales tax is 5.6% is applied. The customer's name and the number of pounds of coffee purchased will be input using an input dialog. Output will be to standard output, System.out.

Program: Open Coffee.java

Step 1: Enter this code into a file and save the file with the name Coffee.java. Because a JFrame object is not created, the input dialog does not have a container. Therefore, the Java reserved word null is passed to the showInputDialog method indicating that there is no parent container.

```java
import javax.swing.*;
class Coffee{
    public static void main(String[] args){
        String name;
        name = JOptionPane.showInputDialog(null, "Enter your name");
        String poundStr = JOptionPane.showInputDialog(null, "How may pounds of coffee would you like?");
        System.out.println("Customer: "+ name + 
        "Pounds ordered: " + poundStr);
    }
}
```

Compile the code and run the program. Record the results.

Reading a method header

Many programs expect the user to enter a number. The showMethodDialog method always returns a String. Since we need to multiply the number of pounds of coffee by the price, we need a way to convert String poundStr to an int,

The header of a method contains the following information

```
<optional modifiers> returnType methodName(<optional parameters list>)
```

The `Integer` class contains a method with header

```
public static int parseInt(String s)
```

- The name of the method is `parseInt`
- A `parameter` is a variable. The parameter list consists of one parameter, named `s`, of type `String`. To use the method correctly, a `String` argument must be passed to the method. The name of the parameter, `s`, is used inside the method and, therefore, does not affect the user of the method.
The return type is int. Thus, the method returns an int value to the user. A method that does not return a value has return type void: the method main is one such method.

The modifier static tells us that the method is a class method, not an instance method.

The access modifier public tells us that everyone has access to the method.

Dot the name of the method with the name of the class in which it is defined, since it is a class method, and pass the method a String argument.

Integer.parseInt(poundStr)

Since the method returns an int that we need to use in a later statement, the returned int is assigned to a variable of type int. Therefore, the complete statement is

int pounds = Integer.parseInt(poundStr);

**Step 2:** Modify the existing code by inserting the new code as shown.

```java
String poundStr = JOptionPane.showInputDialog(null, "How may pounds of coffee would you like?");
int pounds = Integer.parseInt(poundStr);
System.out.println("\nCustomer: " + name + "\nPounds ordered: " + pounds);
```

Compile the expanded code. Run the program and record the result.

---

Note: The java.lang.Double class has a similar method

```java
public static double parseDouble(String s)
```

**Performing calculations**

Let’s continue with this example and calculate the price of an order for bulk coffee beans. Suppose the coffee costs $8.99 per pound. In addition, there is a 5.6% sales tax. On completion, the program should print the

```
Customer: Joe
Pounds ordered: 5 @ $8.99 per pound
Subtotal: $44.95
Tax @ 5.6%: $2.52
-----------------------------------
Total: $47.47
Thank you for your order.
```

First, decide which additional variables are needed:

- a variable to store the cost of the coffee, the value 44.95 above: double costOfCoffee
- a variable to store the calculated tax: double tax
- a variable to store the total cost (cost of the coffee + tax): double total

**Step 3:** Add, after the statement String name; , a single statement that declares all three variables
double costOfCoffee, tax, total;
In addition, the values of the price per pound and the tax rate should be stored as constants to make the code more readable and easier to modify when these values need to be changed. Constants are generally defined at the very top of the code. Add, as the first statements inside the main method

```java
final double PRICE_PER_POUND = 8.99;
final double TAX_RATE = 0.056;
```

In addition, let's begin to write the statement that prints the result. This way, we can begin to format the output.

```java
System.out.println(
        "\n    Customer: " + name +
        "\nPounds ordered: " + pounds + " @ $" + PRICE_PER_POUND +
        " per pound");
```

Your file should now contain

```java
import javax.swing.*;
class Coffee
{
    public static void main(String[] args)
    {
        final double PRICE_PER_POUND = 8.99;
        final double TAX_RATE = 0.056;

        String name;
        double costOfCoffee, tax, total;

        name = JOptionPane.showInputDialog(null, "Enter your name");
        String poundStr = JOptionPane.showInputDialog(null,
                                                "How may pounds of coffee would you like?");
        int pounds = Integer.parseInt(poundStr);
        System.out.println(
                "\n        Customer: " + name +
            "\nPounds ordered: " + pounds + " @ $" + PRICE_PER_POUND +
            " per pound");
    }
}
```

Compile the modified file. Run the program. Record any problems.

_____________________________________________________________________________________________
_____________________________________________________________________________________________

Step 8: Now add the statements that perform the calculations. Add the three statements

```java
costOfCoffee = pounds * PRICE_PER_POUND;
tax = costOfCoffee * TAX_RATE;
total = costOfCoffee + tax;
```

before the print statement. This allows us to keep all of the print statements together.

Compile the code.

After the new code compiles, continue by adding following an additional print statement. Print statements must often be modified several times to produce good looking output. You may use several print statements or one print statement. The choice is yours.
Add these statements to the end of the main method.

```java
//TAX_RATE * 100 is the tax rate as a percentage
System.out.println(
    "\n    Subtotal: $" + costOfCoffee + 
    "\n    Tax @ " + TAX_RATE * 100 + "%: $" + tax + 
    "\n    -----------------------------" + 
    "\n    Total: $" + total);

Compile the program. Run the program and record what is printed.

I ran the program and ordered 8 pounds of coffee. The result was

Order for: Marian
Pounds ordered: 8 @ $8.99 per pound
Subtotal: $71.92
Tax @ 5.6000000000000005%: $4.02752
-----------------------------------
         Total: $75.94752

The tax rate of .056 * 100 cannot be stored exactly in base 2. The inaccuracy occurs within the first seventeen digits and, is therefore, shown in the printout. The numbers 8.99, costOfCoffee, tax and total also cannot be stored accurately in binary. However, for these numbers, the inaccuracy does not occur within the first seventeen digits, so the trailing zeros are not printed.

Using a DecimalFormat object

The printout would look nicer if the number of places to the right of the decimal point was controlled. The Java API provides a class `java.text.DecimalFormat` that is used to format floating-point values in strings for printing. When creating a `DecimalFormat` object, a `String` argument is passed that defines the formatting. For example, the statement

```java
DecimalFormat df2 = new DecimalFormat("0.00");
```

creates a `DecimalFormat` object that formats decimal numbers with two places to the right of the decimal point.

To use the `DecimalFormat` object, use the instance method

```java
public String format(double n)
```

The method `format` has one parameter of type `double` and returns a `String`. Notice that this method is not modified by `static`, and is, therefore, an instance method that must be invoked on a `DecimalFormat` object. The method call

```java
df2.format(3.5)
```

returns the String "3.50". And the method call

```java
df2.format(45.342)
```

returns the String "45.34".
**Step 9:** Immediately after the previously added calculations, add code that declares and creates two DecimalFormat objects.

```java
DecimalFormat df1, df2;
df1 = new DecimalFormat("0.0");  // to format one place to the right of the decimal point
df2 = new DecimalFormat("0.00");  // to format two places to the right of the decimal point
```

And, add this import statement at the beginning of the file.

```java
import java.text.*;
```

Compile the code. Now, modify the last print statement. For each value that should be formatted, pass the value as an argument to the `format` method, using the appropriate DecimalFormat object. For example, in the print statement, the expression

```java
costOfCoffee
```

should be replaced with

```java
df2.format(costOfCoffee)
```

The last print statement should now look like this:

```java
System.out.println("\n Subtotal: $" + df2.format(costOfCoffee) +
    "\n Tax @ " + df1.format(TAX_RATE * 100) + "+%: $" +
    df2.format(tax) +
    "\n-----------------------------" +
    "\n Total: $" + df2.format(total));
```

Compile the code. Run the program, entering 8 pounds, and record the results.

_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________

Compare your results to my results on the previous page. Does the format method round the decimal number or truncate it? List examples from the two outputs to support your answer.

_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________
_____________________________________________________________________________________________

---

**Add Coffee.java and two runs of the final program to Lab04.txt**

**Lab04.txt is now complete.** If you do not have enough time to finish the next two sections during the lab period, you may finish these sections later.
4.5 INVESTIGATE: DECIMALFORMAT

Program: FormatTest.java  Read the code for understanding.

This program investigates some of the different types of String arguments that can be passed to the constructor of a DecimalFormat object and their effects on different decimal numbers. For more information, read the textbook.

```java
import java.text.*;
import java.util.*;

class FormatTest
{
    public static void main (String[] args)
    {
        Scanner scanner = Scanner.create(System.in);

        //The print statement leaves the cursor on the same line
        //as the prompt.
        System.out.print("Enter a number to format: ");

        //The nextDouble() parses the String containing a number
        //into a double value and returns it.
        double number = scanner.nextDouble();

        DecimalFormat df1, df2, df3, df4, df5;
        df1 = new DecimalFormat("00.00");
        df2 = new DecimalFormat("0.0");
        df3 = new DecimalFormat("#.#");
        df4 = new DecimalFormat("#,###.#");
        df5 = new DecimalFormat("$#,###.00");

        System.out.println(number +
                             "\nUsing  00.00:  " + df1.format(number) +
                             "\nUsing   0.0:  " + df2.format(number) +
                             "\nUsing    #.#:  " + df3.format(number) +
                             "\nUsing  #,###.#:  " + df4.format(number) +
                             "\nUsing $#,###.00:  " + df5.format(number) );
    }
}
```

Now run the program, using different input values. Fill in the table on the next page for a record of the results. Feel free to try other numbers and to change the formatting strings that are passed to the DecimalFormat constructor. Note, that a 0 forces a 0 to be printed. A # does not.
4.6 MATH CLASS METHODS

The java.lang.Math class contains data and methods that are frequently used in mathematical calculations. All of the methods are class methods and the two constants

    public final static double PI and public final static double E

are class constants. Notice both of the data values above are modified by final (a constant) and static (a class data value).

Some languages have exponentiation operators, generally ^ or **. Java does not. Instead, exponentiation is accomplished using the Math class method

    public static double pow(double base, double exponent)

The expression

    Math.pow(2.5, 2)

returns the double value \((2.5)^2 = 6.25\). Note that both parameters are of type double. But, in the above statement, the first argument is a double and the second is an int. Java will perform an implicit conversion and cast the 2 as 2.0.

The returned value can be used and not saved, or it can be saved in a variable of type double and used in the future. For example, the following all use the method correctly.

```java
    double areaOfSquare = Math.pow(side, 2);
    System.out.println("Volume of cube = " + Math.pow(side, 3));
    double areaCircle = Math.PI * Math.pow(5.4, 2);
    System.out.println("Volume of sphere = " + 4.0/3.0 * Math.PI * Math.pow(radius, 3));
```

The Math class method

    public static double sqrt(double number)

returns the square root of number. Write two statements that print the square root of 10. One should use the sqrt method and the other the pow method.

4.7 POST LAB EXERCISES

Each of the exercises has two options for I/O
4.1 Write an application WindChill.java that reads input from the user and prints out the Wind Chill Index. The user should input the temperature (T) and wind velocity (V). Wind chill is calculated using the formula

\[
\text{windchill} = 35.74 + (0.6215)T - 35.75(V^{0.16}) + (0.4275)T(V^{0.16})
\]

The wind chill should be printed as an integer value.

4.2 Add shipping charges to the program Coffee.java. The base shipping charge is $3.00. For every 2 pounds, an additional $1.50 is added to the shipping charge. Therefore, the shipping charge is $3.00 for one pound of coffee, $4.50 for two or three pounds of coffee, $6.00 for four or five pounds of coffee, etc. Shipping charges are not taxed.

4.3 Write an application Numeration.java that prompts the user for an integer and prints the number as binary and hexadecimal numerals.

The java.lang.Integer class has methods

- public static String toBinaryString(int n) returns a String representation of n as a binary numeral
- public static String toHexString(int n) returns a String representation of n as a hexadecimal numeral

4.4 Write an application Powers.java that prints the square root, square and cube of the numbers between 2 and 9. Example:

<table>
<thead>
<tr>
<th>number</th>
<th>sqrt</th>
<th>squared</th>
<th>cubed</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.4142</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1.7321</td>
<td>9</td>
<td>27</td>
</tr>
</tbody>
</table>

When printing the square root, print four decimal places. This will be difficult to line up exactly, perfection is not expected. Using the tab character '\t' will help.

4.5 Write an application that reads in three integer values, storing the values in three variables of type int. Print the three values, their sum and their average, a double value. Sample output:

The sum of 12, 17 and 35 is 64.
The average is 21.3333

4.6 Write an application that reads in three floating point values, storing the values in three variables of type double. Print the three values, their sum and their average, a double value and the average rounded to the nearest integer. To round the average you may either use formatting or use the Math class method

\[
\text{public static long round(double) which returns the double rounded as an integer of type long.}
\]

Sample output:

The sum of 58.351, 29 and 43.68 is 131.031.
The average is 43.677
Rounded to the nearest integer, the average is 44