The NetBeans IDE

NetBeans is a programming environment that allows you to create Java programs quickly, while keeping all your files organized and easy to navigate. There are other IDE's (IDE stands for Interactive Develpment Environment) out there, but Netbeans is free and created by the creators of Java: Sun Microsystems.

Once you have loaded NetBeans, select the "File" drop-down menu, and then select "New Project". From there, you will select the top-right choice: "Java Application". "Next". Take note of where NetBeans is saving your project. It should be under "My Documents\NetBeans Projects". The default name for your project will be "javaapplication1.Main". Change the Project name to "Intro". **Important**: Now, *uncheck* the choice of "Create Main Class".

Select "Finish". When you start doing your exercises, how you name them will be very important.

You have now set up a project with not files/programs created it. Now, let's create a new .java program source file.

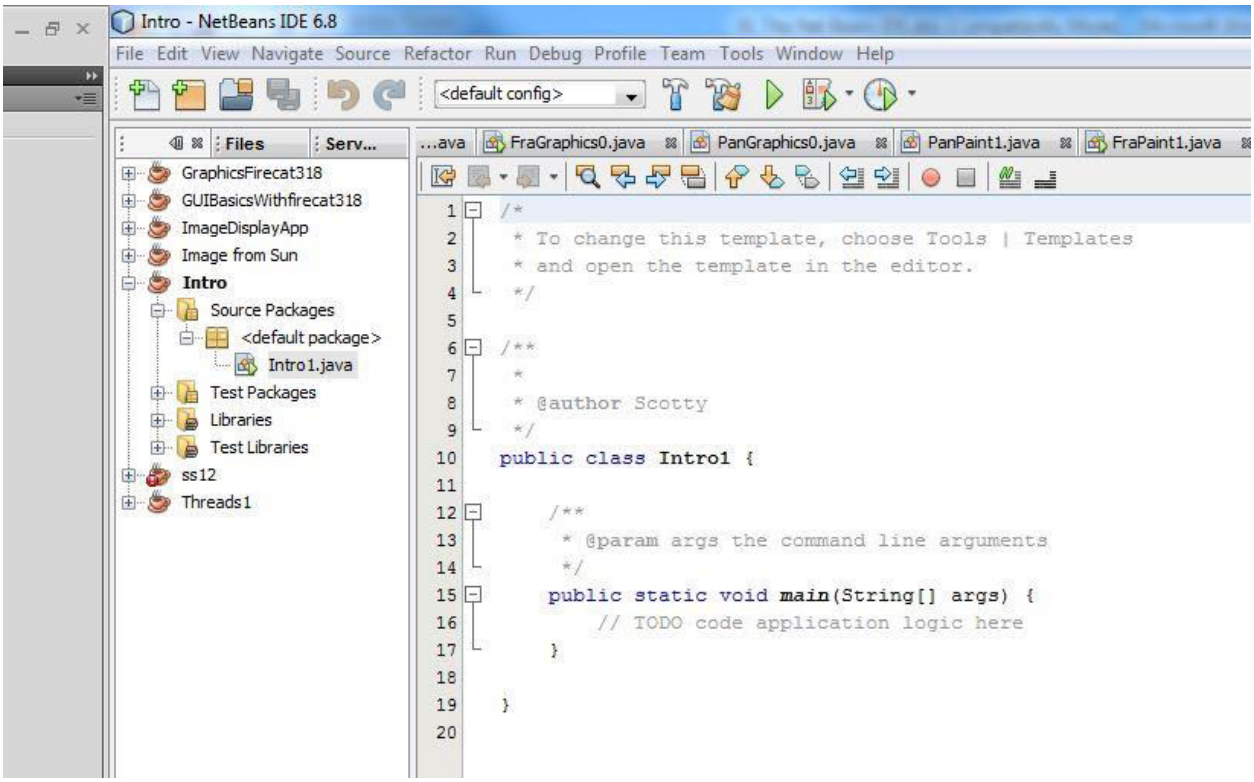
Select "File" from the menu, then select "new file".

Select the "Java" category, and then select "Java Main Class" from the File Types. Select "Next".

Call your new Class Name: Intro1.

Notice that NetBeans is warning you about placing the Java class in the default package. Ignore this warning. Do it anyway. Select "Finish".

Your work environment should look something like this:



Notice that the "Intro" project in the left window is in **bold**. The active project will always be in **bold**. A project holds multiple program files. Under "Intro", I opened up the "Source Packages" folder, then opened the "<default package>" folder, and then you can see the "Intro1.java" program file.

The window to the right (the bigger one) is the source code window. This is where your code goes. Notice that NetBeans tells you where to put the application logic (code) here.

Let's take a look at the code:

First, comments are wrapped between /* and */. That is where NetBeans gives you helpful ideas, and you can write comments to yourself using the same code.

The first line of code is "public class Intro1{". Whatever you named the file when you created it, the first line of code will identify it as a public class. The brace bracket "opens" the class for code. The class is closed with the last brace bracket on line 19.

"main" is within the class "Intro1". It is also opened with a brace bracket. NetBeans then tells you where your code goes. Notice that comments that are on only line only require "//" rather than the open and close comments that I described above.

"main" is closed with the brace brackets on line 17. Notice that since "main" is within Intro1's brace brackets, it is indented. Notice that the //TODO line is within main's brace brackets that it is also indented further.

The program would function properly if this style of indenting was not done, but as you write more and more code, proper indenting becomes very important - especially if you want full marks on a test.

Within "main", NetBeans tells you the following:

// TODO code application logic here

Enter the following:

System.out.println("Welcome to ICS");

Let's entertain the "why" students here.

First, Java is a "case sensitive" language. This code would not work if "System" was typed with a lower case "s".

Secondly, Java is a hierarchical language. That means that that the programming language procedures that you have access to are organized in an object hierarchy. The method "println" (which sends information to the output window) is within the "System.out" object hierarchy.

Thirdly, the println function requires a bracket after it, followed by words within quotations, followed by the close of the bracket. "println" stands for "print line" - after you output, move to the next line. If you want to stay on the same line after you output, use the "print" method.

Finally - all programming statements end in a semicolon. This does not mean "every line". If your programming statement is longer than one line, then continue it on a second line, and then complete it with a semicolon.

Methods:

In skill set 5, you will learn how to create your own methods, but until then, you will be using the ones that Oracle/Sun has created for you. Whenever you call a method in Java, you will need to specify three items:

1. The object that you want to use: System.out in this example.

2. The name of the method: println in this example.

3. A pair of parentheses, containing any "parameters" that the method requires. In this case, we have passed "Welcome to ICS".

You may now run your program using the "play" button. It's the green one at the top that looks like a CD "play" button. Within your output window at the bottom of screen, you should see the following:

```
init:
deps-jar:
Created dir: C:\Documents and Settings\Owner\My
Documents\NetBeansProjects\JavaApplication1\build\classes
Compiling 1 source file to C:\Documents and Settings\Owner\My
Documents\NetBeansProjects\JavaApplication1\build\classes
compile:
run:
Welcome to ICS
BUILD SUCCESSFUL (total time: 1 second)
```

See how it says "Welcome to ICS". There you just did your first Java.

Right below your successful first line of code, enter the word "System" followed by a period. Wait a second. Look at all of the different choices within the System group of code. In the lower window, you can see all of the different choices within “System” to choose. It may seem overwhelming at first, but be happy that the programming environment is prompting you with choices rather than forcing you to memorize them, or *look in a book*. Next, look at the window above. This window gives you a full description of the choice selected in the lower window. Again, this allows you to make informed decisions as to which tool you should use in order to solve the problem. At first, all of these choices and descriptions will look and sound like mumbo-jumbo. Then it starts making sense. Then you start to bore your friends in the cafeteria.

Now enter "System.out" followed by a period. You can see all the different functions that under that. Point down to the one method that we used: the println() method. We used the one that passed a String (more on that later). See when println() is selected, a reasonable description is above it.

Debugging:

If you miss a period or a semicolon, or if you have a capital letter when a small letter should be, things won't work. These types of errors are usually the easiest to fix, but to a beginning programmer, these errors can really be upsetting. If your program is working, try messing around with it. What happens when you delete the semicolon? What happens with you delete one of the periods? Do it NOW while you got it working, to see how helpful the programming environment is when you make these simple, yet frustrating errors.

Project organization:

You will keep all of these programs in one **project** entitled “Skill Set 1”. Call the first exercise e1a, then e1b etc. (Java requires files to start with a letter) DO NOT name your programs any other way!! Your exercises will not be marked if they can not be found, and they will not be looked for.

When you create a new project, call it "Skill Set 1".

Your code will go into the default package folder under source.

Creating e1a.java in Skill Set 1.

Select the "Skill Set 1" folder that is UNDER the "Source Packages" folder. Right click it. Select "New", and the "Java main class". Name the new file "e1a" for exercise 1a.

Once you have your code ready for e1a, you can no longer just hit the "play" button anymore. It will just try to run main.java. **In the project window, select e1a.java.**

Right-click it, and select "Run file". This will ensure that e1a runs. You can also press the F6 button - that runs the currently displayed code.

Why organize your stuff this way?

Officially, all files within a project should coordinate together. That's great for the corporate world, but in the classroom, if we had to create a new project for every exercise, then you would have a monsterous number of folders to navigate. Also, with many exercises within one folder, you will easily be able to navigate your previous work to see where you are screwing up in your current work.

These instructions are repeated in your skill set 1 exercise file.

Variables And Equations

Variables hold information. For example, if you need a variable to store someone's age, then the following will work:

nAge = 15;

Later, you can access the person's age by outputting nAge:

System.out.println("Your age is " +nAge);

That would output "Your age is 15". . Notice that variables are not in quotations. Also take note that the "+" sign separates string constants ("Your age is ") with variables (nAge). The "+" is generally referred to as a "delimeter" in programming terminolgy.

Variable Declarations

All variables must be declared. For example:

int nAge;
char cGender;
String sName;

The first line "declares" that an integer variable will have the name "nAge".

The second line "declares" that a character variable will have the name "cGender".

The third line "declares" a string variable. Strings hold words or sentences.

Java is very demanding with respect to variables. In Java, you are always aware of what "type" a variable is. The act of variable declaration specifies that a variable will hold an integer, character, float, etc. Here's a special note on chars.

Two decisions when declaring variables:

1. What type should it be.

2. What name should you give your variable?

Where to declare?

Declare all of your variables at the beginning of your program. Scattering variable declarations throughout the program is difficult to follow. Declaring your variables at the beginning of the program allows you to tell other programmers what you will be using throughout the program, much like an ingredient list in a recipe.

Initializing variables:
Java allows you to give a variable an initial value upon declaration. This should be done for all variables that entail math. You will experience a lot less errors. For example:
float fArea = 0.0;

Basic equations gets you going with math.

Scope:
as well as the "scope" of the variable. Scope refers to whether a given variable can be used by just that function, or by all functions. The general rule of thumb is you never give a variable more scope than necessary.

Global variables:

Variables that are visible to all functions are declared before "main", or eventually in a "header". Global variables should be avoided whenever possible, since global variables can be modified by any function. This can lead to unintended errors.
Local variables: Variables that are only used within a function, are declared at the beginning of the function, or when a variable is first used.

Type casting:
You may convert values from one type to another, but you may experience some “rounding errors”. For example:
int nNum = 100;
float fNum;
fNum = (float)nNum;

Arithmetic if:
a > b ? c : d If a is greater than b, then c is returned, if not, d is returned.
If you do not want a variable's value to change, preface the declaration with "final". Then, the value will never be modified. If you are going to store the value for pi, why not make it "final" so that you are sure if will never change. Final variables are also known as "constant" variables, since they do not change. Standard naming conventions has a final variable named with all capitals:

final int NMIN = 2;

Identifier:

An identifier is the name of a variable, method, or class. You will learn about the last two later in the course.

Identifiers:

1. Can be made up of letters, digits, the underscore, and the dollar sign. They cannot start with a digit.

2. Spaces are not permitted.

3. You cannot use reserved words.

4. They are case sensitive: nAge is different than nAGE.

Rounding errors and the "float fudge":

A "float" or "double" variable facilitates a "floating point" decimal - values to the right of the decimal place. Try this out:

double dNum = 4.35;

Now output dNum* 100. You should get 434.9999999999994

Why???

If Java cannot convert the value to the right of the decimal place to a number that is over 2 to the power of an integer, it will adjust a "float fudge" in order to make it work. That's why some prefer doubles to floats - it pushes the float fudge factor further down. That's a lot of "f" words.

Variable Types

It is important to know the different types of variables that you can declare. You must learn to properly name them at the time of declaration.

Variable Naming Conventions:

Since you must declare a variable specific type, it is important to name your variables in a way that its type is incorporated in the name. The following is called the "Hungarian" notation, created by Micorsoft's Charles Symonyi. Even though variables would work without adhering to this naming convention, it is great to know, as a programmer what type of information a variable is meant to hold, just by looking at the name. This is a list that will take you through the semester. Initially, you only need to know about ints, floats, char, and doubles. Worry about the other ones later.

Variable Type	Prefix	Size(in bits)	Range	Examples
int	n or i	32	-2b- to 2b+	nAge, nSum
char	c	8	-128 to 127	cGender, cInitial
boolean	b, is	1	true, false	bFinished, isTrue
float	f	32		fQuotient, fCost
double	d	64		dQuotient, dCost
array	ar			arnAge
long	l	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	lNumber

- use "n" as a prefix for an integer, but use "i" to indicate that the integer value is used as an index of an array. (This naming convention came to me as I was trying to make sense of sorting algorithms).
- single lower-case letters are also assumed to be integers used as a counter through a loop

Type Prefix Size in Bits Minimal Range

char c 8 -127 to 127
unsigned char uc 8 0 to 255
int n 16 or 32 -32,767 to 32,767
unsigned int un 16 or 32 0 to 65,535
short (int) n 16 -32,767 to 32,767
long (int) l 32 -2,147,483,647 to 2,147,483,647
unsigned long ul 32 0 to 4,294,967,295
float f 32 six digits of precision
double d 64 ten digits of precision
long double ld 80 ten digits of precision

Oh what type of variable should I use?!
Smaller is elegant - undersized is tragic.
Some prefer using doubles over floats since memory is cheap.
Assigning an unsigned variable a negative number gets very ugly - try it.

Primitive varaibles vs. "wrappers".

Primitive variables are the types that we have been using: int, double, char. These are "old-school" variables that work much like variables have for decades.

Java comes with equivalent "wrappers" for each variable type: Integer, Double, Character. These "wrappers" are fulle "classes" that have many "power-tools" associated with them. You will have a better understanding of them when we work with Object code in a later skill set, but you can still use them without fulling understanding "how" they work.

Old School	Wrapper	Prefix
int	Integer	I
double	Double	D
char	Character	CH

Once you declare a variable of type "Integer", you then gain access to many methods that the Integer wrapper contains. For example:

Integer INum;

Now type INum followed by a period and see all of the methods that you can choose.

Here's a complete list of wrappers:

Primitive type	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Boolean variables:
When you declare a boolean variable, the default value is 1: true for local variables, and 0: false for global variables. It is still better to initialize the variable explicitly by declaring it true or false upon declaration. You should preface a Boolean variable with “is” rather than “b” because it clearly states what the variable is expected to hold.

Escape Codes:

While outputting information, you sometimes need a quick carriage return, or a specific literal that may be needed for formatting purposes. Here's a list:

\n new line

\t tab

\b backspace

\\ backslash - note: in order to get a backslash, you need to enter it twice.

\' single quotation mark (see above)

\" double quoatation mark.

Big Numbers

You will need to import java.math.*; package. Here's the plumbing:

```
BigInteger biNum1, biNum2, biSum;
biNum1 = new BigInteger("100999999999999999999"); // not a really big number
biNum2 = new BigInteger("200999999999999999999");
biSum = biNum1.add(biNum2);

biPrd = biNum1.multiply(biNum2);
System.out.println(biSum);
```

Because BigIntegers are not "primative" or "atomic" variables like int's and double's, you can't use the normal mathematical operators on them. Look at how you need to populate them, add them, or even output them. The BigDecimal class works in a similar fashion.

These "big" variables have essentially no limits on their size or precision.

Characters

Characters:
Characters, or “char’s” hold one single character. To hold more than one letter in a variable, you will have to read my notes on “strings”. You can initialize a letter to a character upon declaration, or get user input to populate the variable. The following code declares the variable as a char and populates it with the letter “d”. Notice the single quotations around the letter “d”.

```
char cLetter = 'd';
What’s the next letter in the alphabet?
cLetter++;
System.out.println(cLetter);
```

ASCII: This definition as well as the copy of the table is from http://www.lookuptables.com/
ASCII stands for American Standard Code for Information Interchange. Computers can only understand numbers, so an ASCII code is the numerical representation of a character such as 'a' or '@' or an action of some sort.
A character variable holds the ascii value of a character at all times. If you make the char variable equal to an integer variable, you will get the ascii value of that character.

Characters are a primitive data type. In order to add some functionality to messing with characters, you can make use of some pre-fabrcated methods. For example:

```
char cLetter = 'a';
char cUpper;
System.out.println(cLetter);
cUpper = Character.toUpperCase(cLetter);
System.out.println(cUpper);
```

See how the "Character" class allows you to act upon a char to return an upper case char.

Chars to ints:

Many times, you are stuck loading numbers into Strings. When that happens, you need to grab individual char's out of strings using the "charAt()" method. Once you have an individual char, then you can convert it to an int. For example:

```
char ch = '5';
int nCh;
nCh = ch - '0';
```

Initially, the char, ch hold the character of '5'. By subtracting the char of '0' from '5', you get the distance between these chars - 5.

Now nCh will have the integer value of 5.

Equations:

A single variable to the left of the equal sign accepts the equation to the right. For example:
*nPrd = nNum1 * nNum2;*

Operators: Mathematical operators are as follows: + - * /

Math and variable types:

When you use an "int", there is nothing to the right of the decimal, because the decimal does not exist for ints. Anything that you may want to the right of the decimal is truncated. Therefore, 8.7 is 8 - there is not rounding.

Division of ints:

Consider the following:

```
int nNum1, nNum2, nQuo; // for quotient
nNum1 – 4;
nNum2 = 5;
nQuo = nNum1/nNum2; // the result will be 0
```

What if you declare the following:
int nNum1, nNum2;
double dQuo; // for quotient
dQuo = nNum1/nNum2; // the result will be 0.0

The reason for this is the following. When you divide two ints, the result is an int. Then the int results will populate a double variable.

Math in the output statement

Please use an extra variable for the result of an equation, and then use a line to calculate it rather than putting it all in the println() method!!! It's just a lot easier for updating and debugging purposes in the future. For example:

```
System.out.println(dNum1/dNum2); // Java permits  but the following is better:
dQuo = dNum1/dNum2;
System.out.println(dQuo);
```

Division by zero:
In Math class, you will learn that a number divided by 0 is “undefined”. In computer class, you will learn it is more than that – it is ugly. Go ahead and try to input 2 numbers and divide the first by 0. Some of the ugliest crashes are sponsored by division by 0 errors. When you review “If” structures, you will learn to ensure that your code never tries to divide by 0.

% **modulus** – this returns the remainder of the division of two numbers: 4%3 = 1
since modulus also uses division, try 4%0 and see what you get.

Postincrement and Postdecrement: I++ and I--.
If n = 2, then n++ is 3. If n is 9 then n-- is 8.
The computer first uses the value to evaluate the expression then increments/decrements the value of the variable.

Preincrement and predecrement: ++I and --I
The computer first increments/decrements the value of the variable and then uses the new value to evaluate the expression.

Try out the following:
```
x = 10;
y = x++;
z= ++x;
System.out.println("x is " + x +" y is " + y + " z is "+ z);
```

What is the value of "y", and why is that so?

Compount Assignment Operators:
Add, then assign: x += 2 This adds 2 to the variable x. For example:
```
x=5;
x+=2;
```
x would now have a value of 7.
```
x-=1;
```
x would now have a value of 6.
```
x/=3;
```
x would now have a value of 2. Got it?

Accumulators:
An accumulator is a variable that "adds to itself". By using the assignment operator (see above), you can accumulate value to an existing variable. For example:
```
x+=n; // accumulates n into x
x-=n; // subtracts n from x
x++; // adds one to x
x--;// subtracts 1 from x
```

It is important to note that you should initialize your accumulators to 0 at the time of declaration. If not, your program will be a mess.

Math in Java

All math functions in Java are under the "Math" class. If you want the square root of a number, you do the following:

double dAnswer;
dAnswer = Math.sqrt(dNum);
dAnswer = Math.pow(dNum, dExp);// this will return dNum to the power of dExp.

When you enter "Math.", look at all of the math functions that pop up. These provide a menu of mathematical opportunities for a programmer.

Fun with trig:

Trig functions in Java take a bit of work; the functions take the angle in RADIANS rather than degrees, so you must convert degrees from degrees to radians before getting any trig ratios.
pi= 3.1415926

Better. The Math class holds pi: Math.PI.
Therefore, the radian value is the value in degrees times pi over 180.
Therefore, the angle will be rad*180/pi

The **Math class in Java** also provides the "ToDegrees" and "ToRadians" functions that do what I described above.

sine: opposite over hypotenuse
cosine: adjacent over hypotenuse
tangent: opposite over adjacent
Math.sin(x); // returns the sine of x(in radians)
Math.asin(x); // arc sine of x (returns radians)
Math.atan(x); arc tan
Math.atan2(y,x); // arc tangent of y/x

Here's a function to convert degrees to radians. Create one that converts radians to degrees.

```
double ToRad(double dDeg, double pi)
{
        return (dDeg) * pi /180.0;
}
dDeg = dRad*180.0/pi;
```

Rounding errors and floating fudge:
People generally attribute computers with precision. Serious precision. Here's the weird part. Digits to the right of the decimal are ESTIMATES more often than not. If the set of digits to the right of the decimal can be converted to a fraction in which the denominator can be converted to 2 to the power of some larger integer, then you are OK, otherwise, some languages like C++ "fudge" the floating point, giving you a meandering "1" somewhere where you really don't want it.

For example, run the following code:

```
float a, b, c;
a = 1.345;
b = 1.123;
c = a + b;
if (c == 2.468)
        System.out.println("They are equal.\n");
else
{
        System.out.println("They are not equal.\n");
}
```

How can you code so that you are not messed over by the floating fudge factor?

1. Don't use floats: use doubles. Double precision floats put the fudge factor further down in relevance. Note: it is still there, but much further down.

2. Don't test for equality of floats or doubles. Do bounds checking with greater than or less than. Testing for equality will usually be false, even if your calculator tells you otherwise.

Bounds Checking: In the above code, substitute the following if structure:

```
dDiff = Math.abs(c- 2.468); // Math.abs() is a function that returns the absolute value.
if (dDiff <0.00001)
        cout&lt&lt"They are equal.\n";
```

Modulus and Random Numbers

Modulus strictly means to return the remainder from the division of two numbers.

For example: 7 modulus 4 will return 3. In code that would be:

nRemainder = 7%4;

nRemainder would then hold 3.

What would nRemainder hold in the following?

5%3
7%3
21%5

Modulus is great for finding out whether a number is odd or even: nNum%2 will return 0 if even.
Modulus is also used to assess if a number is prime or not. (Leave that for a later exercise)
It is also used to restrict a random number to a given range; nRand%10 will be between 0 and 9 - guaranteed.

Random numbers:

The Math class in Java allows you to generate random numbers. For example:

```
int nDie;
nDie = (int)(Math.random()*6) + 1;
System.out.println ("The roll is: " + nDie)
```

The random number generator provides a double number greater than or equal to 0.0 and less than 1.0. By multiplying this number by 6 and then "type casting" this number to an int, you will get a number between 0 and 5. By then adding one to this number, you will get a legitimate value for the roll of one die.

Another way to generate random numbers:

You can import the java.util.Random package, using the following line above your public class line:

```
import java.util.Random;
```

In your code, you can then create a random generator:

```
Random ranGen = new Random();
```

Once created, you can get a random integer with:

```
nRan = ranGen.nextInt(n); // gives you a random integer between 0 (inclusive), and n (exclusive)
```

```
dRan = ranGen.nextDouble(); // gives you a floating-point number between 0 (inclusive), and 1 (exclusive)
```

Output Style

Consider the following three different methods to deal with controlling your output style.

The first:

```
import java.text.NumberFormat;
public class Rounding
{
        public static void main(String[] args)
         {
                double dRad = 5, dArea;
                NumberFormat nf = NumberFormat.getNumberInstance();
                nf.setMaximumFractionDigits(2);
                nf.setMinimumFractionDigits(2);
                dArea = Math.PI*Math.pow(dRad, 2);
                System.out.println("The Area is "+ nf.format(dArea) );
        }
}
```

First, notice the import of the "java.text.NumberFormat" package above the public class.

Second, create an instance of a NumberFormat, and name it "nf".

Third, define nf to have a minimum and maximum decimal of 2.

Next, when outputing the area of the circle, wrap the "dArea" variable in nf.format(), to ensure 2 decimal places.

Second:

This next example converts a decimal to a String in order to round to 2 decimal places:

```
//must import the following:
import java.text.*;
```

Within main you can have the following:

```
DecimalFormat twoDec = new DecimalFormat("#0.00");
double dAmt = 78.456;
//apply the decimal format object to the value you want to format [before outputting]
String sAmt= twoDec.format(dAmt);
//now you can with 2 decimal places
System.out.println("The amount is " + sAmt);
```

First, we create "twoDec" which defines a DecimalFormat as numeric with 2 decimal places.

Next, populate a double variable as 78.456

Then, populate the String, sAmt with the twoDec format. Notice that the formatter rounds the decimal. It doesn't just truncate the value.

The third "old school" method:

System.out.format() allows you to have greater control over how your output is presented.

This is very important for questions that specify your answer be rounded to 2 or 3 decimals. These controls are very similar to the programming language C's printf command.

* d formats an integer value as a decimal value.
* f formats a floating point value as a decimal value.
* n outputs a platform-specific line terminator.

Let's assume that you want a double variable called dNum which holds 45.678 to only display 2 decimal places:

System.out.format("Here's a float %20.2f", dNum);

This is a lot like old C code. Here, your output would start with "Here's a float", and then what happens is the formatting statement that starts with a "%", and then says "allow up to 20 digits, with only 2 decimal places" - that's what "%20.2f" means.

int nNum = 4;
double dNum = 45.678;
System.out.format("Here's an int followed by a float %d is %f.%n", nNum, dNum);

You will get the following output:

Here's an int followed by a float 4 is 45.678000.

The value in nNum is put where the %d is, and dNum is placed where the %f is. This type of formatting is old school.

How about cleaning up the formatting of the float?

System.out.format("Here's a formatted float of %d is %f %<+020.10f %n", nNum, dNum);

Will give you this:

Here's a formatted float of 4 is 45.678000 +00000045.6780000000

Let's rip apart how the formatting took place:

%f %<+020.10f

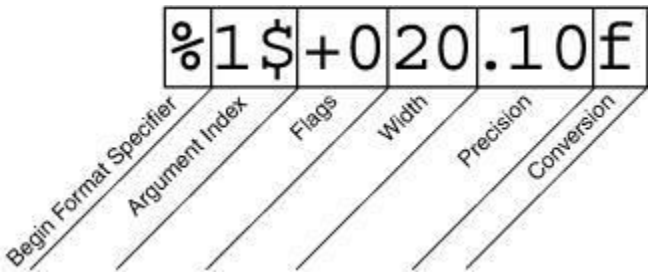
First, the %< refers to formatting the previous number.

+0 refers to the flags. "+" puts sign before the number, and "0" specifies that the number should be padded with 0's to the left of the number.

20 refers to the total length of the number presented.

.10 specifies the floating point precision.

f is to specify a float conversion.



* **Precision**. For floating point values, this is the mathematical precision of the formatted value. For s and other general conversions, this is the maximum width of the formatted value; the value is right-truncated if necessary.
* **Width**. The minimum width of the formatted value; the value is padded if necessary. By default the value is left-padded with blanks.This is total width, including the decimal and the numbers to the right of the decimal.
* **Flags** specify additional formatting options. In the Format example, the + flag specifies that the number should always be formatted with a sign, and the 0 flag specifies that 0 is the padding character. Other flags include - (pad on the right) and , (format number with locale-specific thousands separators). Note that some flags cannot be used with certain other flags or with certain conversions.
* The **Argument Index** allows you to explicitly match a designated argument. You can also specify < to match the same argument as the previous specifier.

Escape Sequences:

Let's say you wanted to output a quotation as part of your output. When you try to do that, the compiler thinks your have completed your "String literal".

System.out.println("The secret word is "Wumbo""); // this will give you a compile error, since Java wants to stop at the quotation before "W".

In order to allow the quotation to be outputted rather than stopping the String literal, you need a "\" before it:

System.out.println("The secret word is \"Wumbo\"");

What if you want to output a "\"? You need a double \\.

New line?? \n is the code for a new line.

Skill Set 1 Exercises

Project organization:

You will keep all of these programs in one project entitled “Skill Set 1”. Call the first exercise e1a, then e1b etc. (Java requires files to start with a letter) DO NOT name your programs any other way!!

When you create a new project, call it "Skill Set 1". When you name your project, you will see a check box in the same window, entitled "Create main class". Make sure this box is **NOT CHECKED**.

In the upper left-hand corner of your screen, you will see the project window. A project holds many folders and files. You will want to go to the "Source Packages" folder, followed by the "Skill Set 1" folder. Here, you will see the "main.java" file. I want your java files to be named better than that.

Creating e1a.java in Skill Set 1.

Select the "Skill Set 1" folder that is UNDER the "Source Packages" folder. Right click it. Select "New", and the "Java main class". Name the new file "e1a" for exercise 1a.

Running e1a.java in Skill Set 1

Once you have your code ready for e1a, you can no longer just hit the "play" button anymore. It will just try to run main.java. **In the project window**, select e1a.java.

Right-click it, and select "Run file". This will ensure that e1a runs.

Why organize your stuff this way?

Officially, all files within a project should coordinate together. That's great for the corporate world, but in the classroom, if we had to create a new project for every exercise, then you would have a monsterous number of folders to navigate. Also, with many exercises within one folder, you will easily be able to navigate your previous work to see where you are screwing up in your current work.

Skill Set 1: Standard output, and calculations.

Section 1: Output and equations.

a) Populate two varaibles, and output the sum and product of those two variables.

b) Create a program that will populate the value of degrees in Centigrade, and then the program will display both the Centigrade and Fahrenheit values in this fashion: CENTIGRADE = 100 FAHRENHEIT = 212. The formula for this is F=(C*9)/5 +32.

c) Populate 2 integer variables, and the program will display the integer value of the quotient.

d) Populate 2 double variables, and the program will display the double value of the quotient, rounded to 2 decimal places. See Output Style for help.

e) When you first learned to divide, you expressed answers using a quotient and a remainder. For example, if you divided 7 by 2, your answer would be given as 3 r. 1. Given 2 integers, divide the first number by the second number. See the Modulus note for help.

f) Create a program that will populate the radius of a circle, and then the computer will display the circumference.

g) Populate the marks for 4 classes. The program will then display the average of the 4 marks.

h) Review your math notes to find the formula for the slope of a straight line and the distance between 2 points in the Cartesian plane. Calculate the slope of any straight line by populating the coordinates of two points in the form of (x,y). Using the same points, find the distance between them.

i) Area and perimeter of a rectangle.

You will populate the coordinates of the upper left corner of a rectangle, and the lower right corner of a rectangle.

Your program will then display the area and the perimeter of the rectangle.

Use as many variables as you need in order to add clarity and efficiency to your code.

j) Quadratics:

Solving Quadratic equations.

The standard form of a quadratic formula is as follows:

ax^2 +bx +c = 0

In order to solve it, the formula is:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},$$

Your program will populate the variables that represent a, b, and c upon declaring them.

Your program will display the two answers: using plus, then the minus before the square root sign. Your answer should be rounded to 2 decimal places.

Use variables that will allow your equation to be broken up into smaller chunks in a way that can add clarity.

Structure: The Art of Being Understood

Structured Programming:
Programming is as much a form of communication as it is a problem-solving tool. Even if your program "solves the problem", you may not get full marks if your programming code is difficult to read. In industry, when you are working on a programming project, your boss is planning on you leaving your position. Hopefully, you will be promoted, maybe you will be fired, or maybe you will find another, higher paying job. The most important part of your program is how readable it is to others.

How do I make sure my program is "structured"?

- Proper names for objects:
- Proper prefixes for objects.
- Proper capitalization of object names and variable names.
- Clear names: long enough to describe, short enough so that you're not in a typing class.
- Indenting when necessary.
- Comments throughout your program to describe what you are trying to do.
- Descriptive variable names with proper prefixes that describe the data that the variable will hold.
- Declaring all variables with the proper scope. There are very few reasons to have global variables.
- NO USELESS PROGRAMMING CODE.
- Using concise programming techniques.

The essence of a good program is as follows:

- **Reliable**: You program can anticipate and handle all types of exceptional circumstances. (See little brother note)
- **Flexible**: The system should be easily modified to handle circumstances that may change in the future.
- **Expandable and reusable**: If the system is successful, it will frequently spawn new computing needs. We should be able to incorporate solutions to these new needs into the original system with relative ease.
- **Efficient**: The system should make optimal use of time and space resources.
- **Structured**: The system should be divided into compact modules, each of which is responsible for a specific, well-defined task.
- **User-friendly**: The system should be clearly documented so that it is easy to use. Internal documentation helps programmers maintain the software, and external documentation helps users use it.
- **Robust**: if a program is completely protected against all possible crashes from bad data and unexpected values. This attribute is really not part of the definition of “structured” but I thought it was important to address here.

What about comments??

Comments on comments:

- Not long prose, just brief notes
- Explain high-level functionality: ++i; /* increase the value of i by */ is worse than useless
- Explain code trickery
- Delimit & document functional sections
- As if to someone intelligent who knows the problem, but not the code
- Anything you had to think about
- Anything you looked at even once saying, "now what does that do again?"
- Always comment order of array indices