

CS 302 – Chapter 2

This week's Goal: to be able to write a complete Java program from memory

Example to refer to:

```
import java.util.Scanner; //Scanner lets us read input from the keyboard

//calculate the product of three integers given by the user
public class Product
{
    /*
     * This method reads three integers from the user and prints their product
     */
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);

        int x; //first value given by user
        int y; //second value given by user
        int z; //third value given by user
        int result; //the product

        System.out.println( "Enter first integer: "); //prompt for input
        x = input.nextInt(); //read first integer from keyboard

        System.out.println( "Enter second integer: "); //prompt for input
        y = input.nextInt(); //read second integer from keyboard

        System.out.println( "Enter third integer: "); //prompt for input
        z = input.nextInt(); //read third integer from keyboard

        result = x*y*z; //calculate the product of the three input numbers

        System.out.printf("Product is %d\n", result); //print the result
    } //end main method
} //end class Product
```

Explain the example:

- This example is what I will refer to throughout class
- It contains things that are probably new to you; don't worry, we will get to those things
- One thing about programs we have not yet said. Look at the use of white space. I have things tabbed over so you can see what is part of the method, the main is part of the class, ect. This becomes even more important in more complex programs. Also note the fact that I left empty lines between code rather than typing them all next to each other. This is important so that your code can be read by people and understood. Also it will be considered when grading your programming projects.
- Also I have shown another way to write comments, especially useful if you have multiple lines of comments together.

-Remember we have a class declared, then a main method declared, and then the body of the main method which says what we want this program to do.

A. Fundamental Data Types

- a. Java already has them so you don't have to write the code yourself
- b. Some of them are objects (like String-recall classes start with a capital letter so String must be an object)
- c. The number types (like integers and double for decimal numbers) are not Objects (you write just int with a lower case letter) but they have Classes. The computer automatically creates the Objects so you don't have to deal with them yourself. This is called "autoboxing" and will be discussed in more detail later.
- d. When you use a number like 2 or 5.6 in Java it is called a number "literal", a String like "hello" is called a String "literal." Literal just means it has a constant value (5 is always 5).
- e. Primitive Data Types (there are 8)
 - i. int
 1. An integer (a whole number) like 12, -5, or 0
 2. Note that real number that is a fraction like the rational number $\frac{4}{3}$ is NOT an int
 3. 32 bit signed number
 4. Uses 2's complement to store it (leading digit shows sign. A 0 means positive and a 1 means negative) This convention makes it easier for the hardware to test if the number is positive or negative
 5. Range (due to only having 32 bits to store it) from -2,147,483,648 to 2,147,483,647
 - ii. long
 1. an integer value but has a larger range than int
 2. You must write it as 7L for the number 7 as a long
 3. 64 bit signed number
 4. Uses 2's complement
 5. Range 9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
 - iii. short
 1. an integer value with a smaller range and smaller to store so saves memory if needed
 2. 16 bit signed number
 3. Uses 2's complement
 4. Range -32,768 to 32,767
 - iv. byte
 1. an integer value with a smaller range
 2. 8 bit signed number
 3. Uses 2's complement
 4. Range -128 to 127
 - v. double
 1. A decimal number
 2. 64 bit IEEE double precision floating point
 - a. IEEE has positive and negative values, also a positive and negative 0, a positive and negative infinity, and a not a number

(NaN) value. Used, for example, if you divide by 0 which is not allowed.

3. Can be written 1.8
 4. Can be written in scientific notation
 - a. 1E4 is 1×10^4
 - b. 5.3E-3 is 5.3×10^{-3} or 0.0053
 5. In a real program you wouldn't use double or float for precise values, you need to use the BigDecimal class that Java provides. But for now, we will just use the double and float.
- vi. float
1. a decimal number, smaller range than doubles and saves memory if needed.
 2. 32 bit IEEE single precision floating point
 3. Must write it as 8.5f if you want the number 8.5 as a float
- vii. boolean
1. Boolean notation is either true or false (sometimes use 0 and 1 in science)
 2. We type true and false in Java
- viii. char
1. characters including a character version of numbers (1) but you can't do math with them, and letters
 2. 16 bit Unicode characters (see appendix A in your book for some of the Unicode and the characters they represent)
 3. Range '\u0000' (0) to '\uffff' (65535)
- f. String class
- i. Not a primitive data type, but also important and we will cover it in this chapter.

B. Variables

- a. Definition: a storage location in a computer program with a name and a value.
- b. The computer automatically links the variable's name with an address in the computer's memory. This is the nice thing about a high level language: you don't have to care where the variable is stored, you just use its name.
- c. A variable can hold only one value at a time, but you can change the value throughout the course of your program.
- d. You can think of variables in CS as a more complex version of variables you are used to in math.
 - i. Example: suppose we have the algebra equation $x+2 = 5$
 - ii. The variable here is x
 - iii. It has only one value (clearly it is 3, this one is simple)
 - iv. Main difference is in algebra you try to find out what the variable's value is while in CS we assign the value for each variable.
- e. Declaration of a Variable
 - i. When we create a variable, we call it declaring the variable
 - ii. Look at the declaration of result in the example (int result)
 - iii. Variable names, when not objects, are lower case by convention
 - iv. First we write its type, in this case int, and then its name
 - v. Often, we have to declare an initial value
 1. Ex: if I wrote `int result = 0;` instead

2. Then I can change the value of result once we have more information. (as we already do in this program)
 3. Note that if you declare it type int, you cannot give it an initial value like "hi" which is a String, not an int.
 4. If you declare an initial value, then we say that the variable result is "initialized" to the value you declared. (0 in the example above).
- vi. You cannot use a variable until you have declared and initialized it.
1. Ex: if I try to add after declaring x, y, and z but before getting the values from the keyboard the line result=x*y*z, the compiler will complain that x, y, and z have not yet been initialized so it doesn't know what values they represent.

f. Variable Assignment

- i. After we have our variable (possibly initialized to an initial value, sometimes it is ok not to) we want to assign a value to it.
- ii. Assignment definition: an assignment statement stores a new value in a variable, replacing the previously stored value.
- iii. Example is the line result = x*y*z. That is where we assign the value of result to be the product of the three numbers we were given. If we had initialized it to 0, then the 0 is replaced by the value of the product.
- iv. We use the "=" operator to do this, as we have seen. Note that = does not denote mathematical equality (for that we must type == but we do not know when we would want to use this yet). = is used to assign values, so it means "is."
- v. Important point to note: the left hand variable is given the value on the right side.
 1. Obvious example: x = 12; Clearly x is assigned the value of 12.
 2. Less obvious example: x = y; In this case the value of y is copied into the variable x. Not vice versa.
- vi. We can do something like this
 1. counter = counter +1;
 2. In this case the current value of counter is incremented by 1 and then counter is given this new value instead. So say counter == 4 right now. After this line of code, counter will be 5.
 3. This may be desired when you want the computer to repeat a sequence of steps a specific number of times. You could make a counter variable and then each time you repeat, increment its value to know how close to done you are.
 4. This has a shorthand notation because it is often used (especially in loops-recall our psudo code with 'while' in it from Friday. We will learn loops in chapter 4).
 - a. counter ++; // counter = counter +1;
 - b. This does the same thing, counter's current value is incremented by one and then stored back in the counter variable.
 - c. counter --; does the same but it decreases the value by one.
- vii. We can do tricky assignments that also do some math
 1. Example: result += x;
 - a. This is shorthand for result = result + x;

2. Example: `result *= x;`
 - a. Shorthand for `result = result*x;`
- g. Variable names
 - i. Make sure it is a clear name that explains what the variable does (example: `result`). Do not use single letters all the time (example: `r`), except perhaps in an algebra solving method.
 - ii. Rules for how to make a name
 1. Lower case first letter for the name of a var.
 2. Begin with a letter or the `_` underscore character
 3. No other symbols (like `?` or `%`) allowed
 4. If the name is two words, such as `studentName`, put the second word with a capital letter so it is clearer, without a space in between them
 5. Case sensitive
 6. Cannot use reserved words (`class`, `double`) because those words already have a specific meaning in Java. See Appendix C for a longer list.
 - iii. Are these names ok?
 1. `String #1Job` (no because of `#` char)
 2. `double product` (yes)
 3. `boolean correctValue?` (no b/c of `?`)
 4. `String 5thGradeTecher` (no b/c can't start with number)
 5. `int _myID` (yes, can start with `_` but not done much)
 6. `BaseballScore double` (no, `double` is a reserved word)
- h. Literals vs variables
 - i. Variables must be declared
 1. `String name = "Alicia"`
 2. `int counter = 5;`
 - ii. Literals are actual Strings or numbers
 1. `"Alicia"`
 2. The `2` in the formula: `int dozen = 10+2;`
 - iii. So in the first example, `name` is a variable with the value `Alicia` and `"Alicia"` is a string literal. `Counter` is a variable with a value `5` currently but `5` is a literal.
- i. Constants
 - i. A variable whose value can never change which you need in your program
 - ii. By convention you write its name in all caps with `_` in between words, to help your readers keep it and normal variables separate in their minds
 - iii. Declared with `final`
 1. `final double MASS_OF_EARTH = 5.97E24 //5.97 x 10^24 kilograms`
 2. `final double LITER_PER_OUNCE = 0.0296;`
 - iv. Declare it rather than typing the number into the formula every time you need it so those not familiar with the formula you are using don't wonder why this number is here
 1. `Double numberOfEarthsInSun = MASS_OF_SUN/MASS_OF_EARTH;`
 2. Above makes more sense than just having two huge numbers divided by each other.
 3. `Double canVolume = canOnces * LITER_PER_OUNCE;`
 4. `Double canVolume = canOnces*0.0296;`
 5. `3` makes more sense than `4` to your readers.

- v. We call these numbers as in 4 above, where you don't know why they are there 'magic numbers.' Avoid using magic numbers.
- vi. Another reason to do this is that if the constant changes for some reason, you can just change what the final var. is set to, rather than finding every spot in your code that number was typed and changing them all.

C. Reading Keyboard Input

- a. The import line above the class declaration is because we have to tell Java that we want to use the Scanner class, which reads input.
 - i. Scanner is in its package until we import it. (it is in java.util hence why we wrote that)
 - 1. Package is a collection of classes for a related purpose
 - 2. Java library is full of packages
 - ii. The System class (as in System.out.println()) is in the java.lang package which is automatically imported into all of your programs)
- b. We create a Scanner type object called input so that we can use the Scanner methods. (more about this when we do objects later)
- c. First line printed in example is a prompt: we always have to tell the user what we want before we take input from keyboard
- d. Then we use the Scanner class's nextInt() method, which waits for the user to type in an integer number and hit enter. We have to type input.nextInt() because input is a Scanner object, so it has permission to use the Scanner methods. You can't just type nextInt() because the method is not defined in your class.
 - i. Note: the nextInt() reads in the int that was typed, but it leaves the return character, from when the user typed enter, sitting on the input line. This can cause problems when you read in the next value so we sometimes add an input.nextLine() after the input.nextInt() to get rid of the return character.
- e. There is also a nextDouble() and other methods.
- f. You can also read in a string from the console using next()
 - i. Note that if more than one word is typed, this will only read one of the words
 - ii. You will need another input.next() in your code to get the second word and so on.
- g. If you use nextLine() this will read in the whole line that was typed as a String.
- h. You can find out about all the Java library classes and methods using the API documentation as described on page 43 of your text.

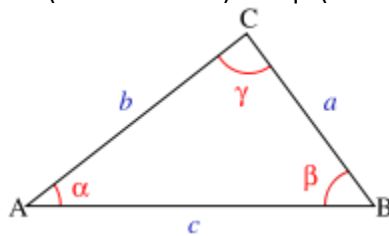
D. Arithmetic

- a. Table I took from another book on arithmetic

Operation	Operator in Java	Example in Math	Example in Java
Addition	+	f+7	f+7
Subtraction	-	p-c	p-c
Multiplication	*	bm	b*m
Division	/	x/y	x / y
Remainder (Modulo)	%	r mod s	r % s

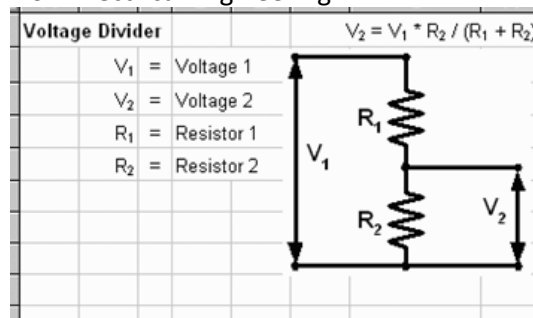
- b. Be sure to use these symbols. Ex: cannot write 5x6 to mean multiply, because Java won't understand that.

- c. Expression-as in math, a combination of variables, numbers (literals), and operators or method calls.
- d. Same precedence rules as in math
 - i. Ex: $12-4/4$ and $(12-4)/2$ are not the same
 - ii. Use parentheses whenever needed by math or to make things clearer
 - iii. Rules of precedence
 1. Multiplicaiton, Division, and Modulo happen first
 2. Addition and Subtraction are next
 3. If both from one level exist, then left to right
- e. How do we write these formulas in Java?
 - i. From physics (motion under constant acceleration): $s = s_0 + v_0t + 1/2gt^2$
 1. `double s = s0 + v0*t + .5*GRAVITY*Math.pow(t,2);`
 2. GRAVITY = 9.8 meters per second declared earlier in program
 - ii. From math (law of cosines) $c = \sqrt{a^2 + b^2 - 2ab\cos(\text{gamma})}$



1. `double c = Math.sqrt(Math.pow(a,2) + Math.pow(b,2) - 2*a*b*Math.cos(g));`

iii. From Electrical Engineering



1. `Double v2 = v1*(r2/(r1+r2));`

- f. We talked about int and double being different and how you cannot write `double x = "hi."`
 - i. One place that does not apply is the computer can make an int into a double easily, it just adds .0 to the number.
 - ii. `double number = 5;` is ok
 - iii. Note that if you try to make a double be an integer, you lose all the info that was in the decimal part and it is possible that the double is outside the range that an int can store.
 - iv. `int number = 4.5;` is not ok
 - v. Also, if you do math with ints and floating point numbers, a floating point value will be computed
 1. $7*2.0$ will be 14.0, not just an int
- g. Suppose you really want to make a double become an int (or do the same with some other Objects). The way we do this is with a "cast."

- i. In essence, you are telling the computer that you take responsibility for any loss of information
 - ii. Normally, do not cast things unless you are sure that there will be no info lost (you have a double value that you know is actually 6.0), or you don't care about the info lost for some reason.
 - iii. example:


```
double balance;
int dollars = (int) balance;
//here we will throw away the decimal part of the balance, leaving only
the number of whole dollars. If balance 20.45 then dollars is 20.
```
 - iv. Note that if you do not want to lose the fractional part, you can use a Math function. If balance is 12.75 then the top one would give 12 and the following would give 13.
 1. `int dollars = (int) Math.round(balance);`
- h. Integer Division
- i. Easily could have a test question that involves this
 - ii. Remainders are discarded
 - iii. $8/2 = 4$ and that is fine
 - iv. $7/2 = 3$ because the remainder of 1 was thrown away
- i. Modulo and Remainder
- i. You can do math that needed modulo notation
 - ii. $5 \bmod 4$ would be `5 % 4`
 - iii. This is essentially the same as just saving the remainder
 - iv. $7/4$ is 1 remainder 3, so `7%4=3`
 - v. There are uses for this, but they are mostly very specific
 - vi. One general example from text: say you have a pile of pennies and want to know how much money you have
 1. `int pennies = 1729;`
 2. `int dollars = pennies / 100; //17`
 3. `int cents = pennies % 100; //29`
- j. Advanced Math things
- i. Mostly use the Math class for this (see API documentation to find out all of the different methods there are. See page 43 to get the webpage to use.)
 - ii. `Math.sqrt(n)`
 - iii. `Math.pow(a,b) = a^b`
 - iv. `Math.PI` is the number pi
- k. Table on operators

Algebraic operator	Java operator	Java example	What it means
=	=	<code>x == y</code>	x is equal to y
≠	<code>!=</code>	<code>x != y</code>	x is not equal to y
>	>	<code>x > y</code>	x is greater than y
<	<	<code>x < y</code>	x is less than y
≥	<code>>=</code>	<code>x >= y</code>	x is greater than or equal to y
≤	<code><=</code>	<code>x <= y</code>	x is less than or equal to y

E. Strings

- a. Definition: a sequence of characters (including letters, numbers, punctuation, spaces, etc)
- b. Length of the string is defined as the number of characters in it. After you declare a String variable you can get the length with `name.length()`;
- c. You can use an empty string `""` as a default, which has a length of 0.
- d. String is an object, so it must be capital when you declare String variablesString
 - i. `String name = "Alicia";`
- e. Concatenation
 - i. Two strings (either variables or literals) can be put together to form one string with the `+` sign.
 - ii. This also works if you concatenate a String and a number, because the number is changed into a string and then they are concatenated. You can also concatenate a String and a variable.
 - iii. This is used a lot with print statements
 1. `System.out.println("The result of this program is: " + result);`
- f. We learned how to change doubles into integers before, we can also change Strings into integers. Using the `Integer.parseInt()` method
 - i. Example: `String number = 45; int nu = Integer.parseInt(number);` then `n` becomes 45
 - ii. If you try to do this with a non-integer number (like `"45.9"` or `"cat"`) you get a run time exception.
 - iii. There is also a `Double.parseDouble()` method to read doubles from Strings.
 - iv. If you are wondering what this Double and Integer are, and you think with capital letters they look like objects, you are correct. These are the classes I mentioned once before that Java secretly creates every time you make an int or double.
 - v. You might have also noticed that we did not declare any Objects named `"Double"` here unlike all the other Object examples we have done (Scanner). That is because `parseDouble()` is a static method, which as I mentioned means that there is not a different version of it for each Double object created. That means you don't need to have a specific Double object to run it (unlike in our example of Student objects, you need a specific Student object to ask what its name is), because it exists for the entire class.
 - vi. It might also occur to you that methods like `Math.sqrt()` that we discussed on Friday are also static and this is why you don't have to declare a Math object before you use the method.
 - vii. For static methods, rather than making an object and typing `ObjectName.methodName()`, you can just type `ClassName.methodName()`
- g. Formatted Output
 - i. See the final print line in the example
 - ii. You put a `%c`, where `c` is some letter depending on what goes there. This is a 'placeholder' for the actual number or string that will be printed to the console.
 - iii. Escape sequences, which you have a table of, can be used in formatted output, like to make a new line, or in String variables if you want quotes or backslashes in them for some reason.

1. You might want the `\` when you want to print out Someone said, "" and then enclose their words in quotes. You don't want java to think your String object is over at the " mark as it normally would
2. You might want a new line to break up a really long message you are writing, or to make a picture on the screen
 - a. `System.out.println(" * \n *** \n*****");` prints:


```

*
***
*****
          
```
3. You might want the `\\` if you are writing a filename into your code

h. Table of Escape Sequences

Escape Sequence	Description
<code>\n</code>	Newline. Position the screen cursor at the start of the next line
<code>\t</code>	Tab. Move the cursor to the next tab stop
<code>\r</code>	Carriage Return.
<code>\\</code>	Backslash. Print out the <code>\</code> character
<code>\"</code>	Double quote. Print out the <code>"</code> character

i. Table of placeholders in the formatted output (ex: the last line in the example)

What you type	What kind of value you output
<code>%s</code>	String
<code>%d</code>	int
<code>%f</code>	double
<code>%.2f</code>	double with two decimal points like 3.45 or 12.34. Note that if you change 2 to a different number, you get that many digits after the decimal point
<code>%c</code>	char

j. characters

- i. They are enclosed by single quotes
 1. `char letter = 'a';`
- ii. They are actually encoded as numbers
 1. See appendix B for their values
 2. `System.out.println('H' + 1);` is 73 because H is encoded as 72
 3. Note that H and h are different chars and are different numbers
- iii. Strings are made up of a bunch of characters, you can pull out the *i*th character of a String with the `charAt()` method.
 1. First, we need to note that we start our counting at position 0. (This will also be true of arrays when we learn about them). But when we count the length of the String it starts counting at 1. So this string has a length of 6, but the characters are labeled `char0` to `char5`. Think of this as, this word clearly has 6 letters, but we always start labeling at the 0th one.

A	l	i	c	i	a
0	1	2	3	4	5

2. `charAt()` is NOT a static method, because it has different results for each String

3. To use a method of the String class on a specific string, you type
ObjectName.method();
4. String myName = "Alicia";
char myFirstLetter = myName.charAt(0);
result of myFirstLetter is A.

k. substrings

- i. We talked about pulling out the characters of a string, but what if I have a String that is first and last name, and now I only want the first name. I could pull out the characters one by one, but that would not make a String object.
- ii. We use the substring() method to do this
- iii. As above, substring is not static, so you must call it using StringName.substring()
- iv. substring() has two parameters: the start point for the substring, and the char AFTER the endpoint for the substring. This is still counted from char0.
String myName = "Alicia Maxwell";
String myFirstName = myName.substring(0,6); (this gives "Alicia"
because it makes a string from char0='A', char1, char2, char3, char4,
and char5='a'. Notice that char6 is NOT in the substring.)
- v. There is also a version of substring() that has only one parameter: the start point for the substring (counting from Char0). Then the entire rest of the string from the start point on is taken
String myName = "Alicia Maxwell";
String myLastName = myName.substring(7); (this gives "Maxwell"
because it takes char7='M' to the end of the String)
- vi. You can of course take only one character into your substring. Since you cannot concatenate characters into a String, this is a way to concatenate a bunch of Strings of length 1 into a String. The text gives an example where it takes the first initials of two people dating and makes a String "R & S" like people carve on trees. I mention this to say why they didn't just use charAt(0) on both names, because you can't concatenate them.

F. Random Class

a. Constructing a Random Generator

- i. Pseudorandom numbers: drawn from a sequence of numbers that does not repeat for a very long time and so appears to be random.
- ii. Zero argument constructor: Random rng = new Random();
- iii. Seeding: Random rng = new Random (long someSeedValue);
 1. Note that to use the number 1 as my seed, I need to type 1L so the computer knows it is of type long, not type int.
 2. The seed is used in the algorithm that computes the pseudorandom number. The zero argument constructor uses the timestamp from your computer at the time you run the program as the seed. Random number generators that have the same seed always generate the same random numbers in the same sequence. Thus if you know what seed you are giving it and which algorithm is being used, then you can always know exactly what pseudorandom number will be generated when

b. Methods of Random

i. `nextInt(n)`

1. Return a random int or floating point value between 0 (inclusive) and n(exclusive)
2. Note that this is written $0 \leq x < n$. This exclusive at the top matters, and make sure you think twice about it in your code to make sure you are doing what you want and not off by one.

ii. `nextInt()`

1. Returns a random number in the range of allowed ints in Java (which is about -2.1billion to 2.1 billion)

iii. `nextDouble()`

1. Return a random double between 0 (inclusive) and 1 (exclusive)

iv. There are other methods that we will not use in this class. Some of them are listed below.

1. `nextGaussian()` returns a random double from a Gaussian distribution, which you need in statistics, physics, and other sciences
2. `nextBoolean()` returns true or false with approximately equal probability
3. `nextFloat()` returns a float between 0.0 and 1.0
4. `nextLong()` returns one of the 2^{64} long values in the range of allowed longs

c. Shifting the values into the desired range

i. Generate an int between 8 and 11

1. `nextInt(4)` will return a value between 0 and not-including 4
2. to get the lower bound to change from 0 to 8, we just need to add 8 to 0, suggesting `nextInt(4) + 8`
3. What will this do to the upper bound? It will turn from 4 (exclusive) to $4+8=12$ (exclusive) so the highest allowed integer will be 11, just as we want.
4. Conclusion: use `int n = rng.nextInt(4) + 8;`

ii. Generate a number in the range of 1-6

1. `nextInt(6)` gives you a number $0 \leq x \leq 5$
2. So if we just add 1 to that, it will move to being a range of $1 \leq x \leq 6$
3. Solution: `int roll = rng.nextInt(6)+1;`

iii. Generate a floating point number between 0 and 2.

1. `rng.nextDouble()` returns a value between 0 and 1
2. Then to change this to a value between 0 and 2, we can just multiply by 2
3. Solution: `double n = 2*rng.nextDouble();`