CSC142 Supplemental Reading: Week02

Contents

1 Class Variables and Methods

1.1 The Math Class

The Math class contains a large number of tools for performing calculations. You will certainly use some of these tools this quarter and in any future programming you do in the Java language. If your past experience is with JavaScript, you will find that many of the tools in Java's Math class are the same as those available in JavaScript's Math object. One difference, however, is that you must be aware of data types when using the Java tools.

One way to become familiar with the Math class is to view its specification in the Java API Documentation. Open this documentation now and browse to locate the Math class (you can search for the Math class in the list of all classes on the left-hand side of the Java API website). Take a few minutes to browse through the Math class specification. Focus on developing a feeling for what kinds of tools are available. Review the list of methods so that you know where to look if you ever need to perform any of these kinds of calculations. In this course, if we ask you on a test to use a method in the Math class, the **method signature** will be provided to remind you about the method. You will be responsible for being able to use the method given its signature. Here are the method signatures of methods you should become familiar with:

```
double Math.abs( double d )
int Math.abs( int d )

double Math.pow( double b, double e )
double Math.sqrt( double d )

long Math.round( double d )
double Math.ceil( double d )
double Math.floor( double d )

double Math.min( double a, double b )
int Math.min( int a, int b )
double Math.max( double a, double b )
int Math.max( int a, int b )

double Math.sin( double d )
double Math.cos( double d )

double Math.log( double d )
double Math.exp( double d )

double Math.random()
```

When using methods in the Math class, consult the documentation to make sure that you understand the following:

- What information does the method require (what are the **parameters**), and what are their **units**? For example, trig methods like sin() and cos() require that all angles be in radians.
- What data types are expected for parameters and what is the data type of the value returned?

1.2 Using Class (static) Methods

The following statement illustrates the syntax for using a method of the Math class:

```
double distance = Math.sqrt( diffX * diffX + diffY * diffY );
```

Notice that when the **sqrt** method is called, the name of its **class** is used as a qualifier. All of the methods in the Math class are examples of **class methods** (or ***static* methods**).

1.3 Named Constants (final variables)

When you are going to use constants in your code that have particular meaning it's good programming practice to use **named constants** instead of entering "magic numbers" where they are used. One way to accomplish this is to create a variable (with a good name) and store the constant in the variable. The one drawback to that is because it is a variable, a method could accidentally change the value (you say to yourself "but I'd never do that!" Consider though that someone else may modify your code or you may just forget...)

To address this potential for disaster (!) any variable in Java may be declared ***final***. A ***final*** variable can never have its value changed after the initial assignment, the compiler will catch an attempt at a reassignment as a syntax error. Therefore, be sure to initialize a final variable on the same line where it is declared.

Named constants are often class (*static*) variables. Here's a statement that declares and initializes a variable of this kind:

```
public static final int MAX_WORD_LENGTH = 1023;
```

Make sure you're clear on what each keyword does: public -- static -- final.

This example also illustrates a naming convention: named constants should be written in all caps with underscore characters separating multiple words in the name.

`Math.PI` and `Color.RED` (see the Color class in the Java API) are two examples of named constants (and class variables) in library classes.

2 Using Strings

One commonly used reference (object) type in Java programs is type **String**. A String is an object that stores a sequence of characters and has many methods that can be used to operate on those characters. Because strings are so important, Java includes some shorthand syntax notation for working with strings.

2.1 Literal Strings

Just as literal numerical values may be entered into a program, Java allows for literal string values to be used. Using a literal string value effectively constructs a new String object. Here's an example -- this statement:

```
String s1 = "Java Rocks!";
```

has the effect of 1) declaring a new String variable **s1**; 2) instantiating (creating) a new String object; and 3) binding the variable s1 to the new object. Notice that we don't have to use the keyword **new** in this case. One of the shorthands provided by the language.

This part of the statement:

```
"Java Rocks!"
```

is an example of a **literal String** value.

2.2 length() and charAt() Methods

A String may be thought of as an ordered collection of characters. In the example above, the String s1 is made up of 11 characters. The quote marks around a literal string are not counted; those characters are not part of the string, but rather serve to mark the beginning and end of the series of characters that *are* part of the string. The number of characters a String contains is referred to as the **length** of the String, and String objects have a **length() method** that can be used to determine this value. So, given the statement above, the expression **s1.length()** evaluates to 11.

The characters that make up a String can be thought of as being numbered, in order, starting with the number 0. (In fact, you will discover that programmers like to count beginning with the number 0.) The character at a particular location in a particular String can be determined by calling that String's **charAt() method**. For example, given the assignment above, the expression **s1.charAt(0)** evaluates to **'J'**. Notice the single quote marks around the 'J' -- that's the way a literal ***char*** is written in Java. Type ***char*** is one of the primitive types, and this is the type returned by the charAt() method. What value is returned by the method call **s1.charAt(4)** ? If you answered **' ' (a space)**, you are correct. A space is treated like any other character in a String, and character 4 is a space.

The argument (actual parameter) passed to the charAt() method must be a valid 'index' -- that is, there must be a character in the location specified. For example, each of these method calls: **s1.charAt(11)** and **s1.charAt(-1)** would result in a **runtime exception (IndexOutOfBoundsException)** because there is no character number 11 in the String s1 (the 11 characters are numbered 0 thru 10) and the number -1 can never be a valid index.

The String class has many other methods that can be used to analyze and manipulate String objects. When the time comes to use them in a future assignment, you can find all the information you need by looking up the String class in the Java API Documentation.

2.3 String Concatenation

In Java, the plus sign (+) when applied to String objects performs an operation called **string concatenation** -- joining two strings together to form a new, longer one. So, the following statement:

```
String s2 = "It's Time to Say " + s1;
```

assigns the following value to the variable **s2**: "It's Time to Say Java Rocks!". If the + operator sees a String object as either of its operands, it will perform string concatenation. This can lead to some minor confusion, as illustrated here:

```
int x = 5;
int y = 4;
String wrongAnswer = "The sum is " + x + y; // produces "The sum is 54"
String rightAnswer = "The sum is " + ( x + y ); // produces "The sum is 9"
```

Immutability

String objects are said to be **immutable**. That means that once created, their **state** (the characters in the string) **can never be changed**. If a string can never be changed, then what happens when a statement like this is executed?

```
s1 = s1 + " my friend";
```

The answer is that the expression **s1 + " my friend"** has the effect of constructing a new String object. This new object is then bound to s1.

3 Java import statement

Packages and the import Statement

When dealing with a large number of classes, it is helpful to organize them in some way. In Java, a group of related classes is called a **package**. Every package (except for the 'default' package) has a name. For example, there is a class named Color that is part of a package named java.awt (if you go to the Java API and look up the Color class, you'll see its package listed at the top its documentation). One benefit of organizing classes into packages is the avoidance on name collisions. For example, there could be another class named Color that is part of another package.

Every class has a **fully-qualified name** that consists of the name of its package followed by a 'dot' followed by the name of the class. So, the fully-qualified name of the Color class is java.awt.Color. The fully qualified name of a class can be used anywhere the name of a class is needed. Of course, coding is easier (and code is easier to read) if we can use *only* the class name (known as the **simple name**). This is possible by using an import statement to direct the compiler to look into a certain package for classes used in a given program. Here's an example -- this code uses the fully-qualified name for the Color class:

```
public class Test {
   public static void test() {
      java.awt.Color c = new java.awt.Color( 20, 40, 60 );
      // more code here
   }
}
```

This code accomplishes exactly the same thing, but makes use of an import statement so that the simple name of the Color class may be used:

```
import java.awt.Color;
public class Test {
   public static void test() {
      Color c = new Color( 20, 40, 60 );
      // more code here
   }
}
```

An import statement may be used to import a single class, as was done above, or to import an entire package, like this:

```
import java.awt.*;
```

The name of a package may include a 'dot', as you see above, but that does not imply that packages are nested. So, a statement like this:

```
import java.*;
```

would NOT import any classes in the package java.awt. The complete name of any package you wish to import must appear in an import statement.

Java automatically imports all classes from the package named java.lang.

4 Functional Decomposition

When you think of the word decomposition, what do you think of? Breaking something down into smaller pieces, perhaps? This idea can be applied to programming as well.

You may have come across the phrases -- functional (or procedural) decomposition or modular programming. They all refer to the same idea: breaking larger tasks into smaller, more manageable pieces. For example, an initial task might be to "generate a report." This might be made up of the smaller tasks:

1. collect data
2. sort it by certain criteria
3. print it to the screen (or file, or printer)

Several benefits result from functional decomposition:

Code is easier to design -- a large task can be broken down into smaller pieces, then each piece can be designed as a separate method.

- Code is easier to write -- Once you have these individual operations designed, you can focus on writing and testing just one at a time, instead of tackling the whole task at once.
- Code is easier to read -- when methods are given meaningful names, the code that makes use of them can be almost self-documenting. In addition, detail can be hidden that distract from the overall meaning. Consider the following example from a proposed bank account management program:

```
var accountNumber = verifyUser( username, password );
var depositAmount = getDepositAmount( );
var newBalance = applyDeposit( accountNumber, depositAmount );
addMonthlyInterest( accountNumber );
```

You can get a high-level idea of what is going on, even without knowing the details of how each of those functions work (also known as procedural abstraction).

- Code is easier to test and debug -- individual, smaller methods can be tested separately before they are combined in a larger method.
- Code can be re-used -- a well-written (general-purpose) method can be reused in many situations, such as Math.pow().

Communication from some portion of code to a method is usually via the parameters and the return statement. Don't make a variable a class variable just to avoid passing parameters.

5 Stepwise Refinement

As you learn more features of Java, your programs are becoming more complex. So the question is, when someone (a client, a boss, a teacher) asks you to write a program how should you get started? As you've seen before, it must start with a plan.

5.1 Planning your Design

If you recall, it is important to get your design ideas down on paper before you code. Just like you want a contractor to have plans to build your house, you don't want to start writing code without a design. The tools we've seen for helping to organize your design ideas are flowcharts, pseudocode, and hierarchy charts.

Now that our programs are getting bigger, there are more details to plan, sometimes too much to think about at once. That's why programmers adopt the **stepwise refinement** or **decomposition** approach. Stepwise refinement attempts to solve a complex problem by breaking down larger tasks into smaller tasks. These subtasks may be broken down into even smaller (or finer) tasks, until the size of the task is manageable and you can easily "see" the solution.

5.1.1 Stepwise Refinement with a Research Paper

Think for a minute how you go about writing a paper. You don't just sit down and start writing, correct? (Hopefully…) What you do is start with an outline. Something like this:

I.
II.
III.
IV.

These are the major topics or sections of your paper. The next step is to go into each section and start to work out the details. You typically focus on one area of the paper at a time:

I.

a.
b.

II.

a.
b.

1.

2.

c.

III.

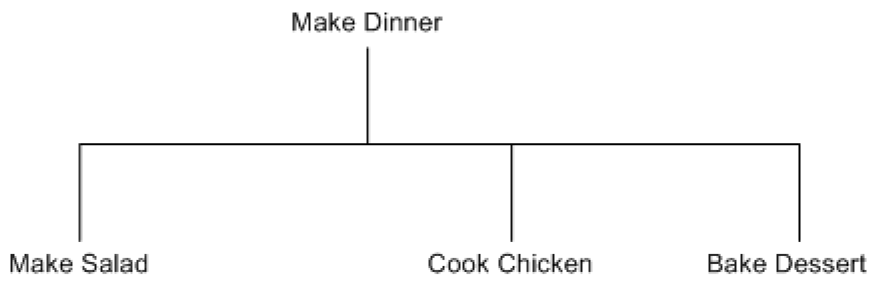
a.
b.

IV.

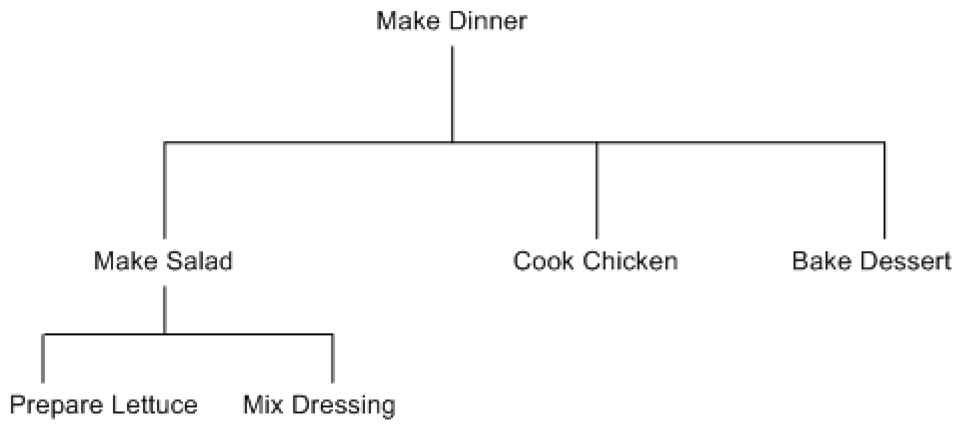
Now, some sections (like section IV) may not need any extra details. You can envision what will go there already. Some sections (like section II)may need a lot more detail (subsection IIb needed to be broken down into even finer details). Once you have this outline done to the level of detail necessary for you to write, then you can start writing the actual paper.

<u>Another Stepwise Refinement Example</u>

Let's say the task is to cook dinner. You first refinement may look like this:



Now, focus on just one step: making salad. What does that entail? Your continued refinement may look like this:



Realize that everyone's refinement may be different. For some, "Mix Dressing" means opening a bottle, no refinement required. Others, who may make the dressing from scratch, would refine this step further.

How about "Cook Chicken?" What steps would go there? See how the process evolves?

5.1.2 Stepwise Refinement for an Algorithm

You should adopt the same approach when working on the design of a program. Break the large task (the entire program) into smaller pieces that you can focus on one at a time. Start with the basic flow of any computer program:

INPUT
PROCESS
OUTPUT

Then, think for a moment just on the input process. What are the inputs? What are the data types? Do you need data validation? You may then update the design like so

```
INPUT

get user's name (string)
get user's grade and validate (integer from 0 to 100)

PROCESS
OUTPUT
```

This was the first step in refinement. Now, getting data is pretty straightforward, no need to refine that further. But what about validating? At this point in your experience, that may require a little more thought. We've learned to use a loop for data validation, so let's refine this step further

```
INPUT

get user's name via a prompt (string)
get user's grade via a prompt (integer from 0 to 100)
while (invalid grade)

ask again

PROCESS
OUTPUT
```

Notice the use of pseudocode here. I'm not focused on syntax, just the logic. There's still more refining to do: putting the actual test in the while loop.

You can see how you would continue this procedure as you move through the PROCESS and OUTPUT sections. Expect the PROCESS section to be the most challenging. That's the place you really want to think about using refinement. Start by just listing the big tasks within the PROCESS section, then focus on solving each task.

5.2 Ideas to Keep in Mind

- When you are working on one section, don't be thinking about another section. Keep your attention focused.
- You don't have to solve everything at once. That's the idea of refinement. For example, try a first pass at refining a section: if a step feels pretty big, just write the step down as one line (such as "DO THIS TASK"), don't worry about the details of that step yet. Then go back to that step and work out its details.
- Don't be impatient with yourself. It takes time to think about how to solve even a small problem. As you gain experience you will get better.
- Write your ideas down. Write your variable names down. Write your formulas down. Write **everything** down! If you are not writing them down and instead are trying to remember it all, you will not be able to focus on the task at hand.
- You may go through different designs the way you go through drafts of a paper. Don't think you have to get it right on the first try. You may end up throwing away some idea because it didn't work. That's ok. Professionals do that too. It's expected.
- Remember, this is pseudocode or flowcharts, not actual coding. Don't get hung up on syntax at this point. That will just distract you from creating the logic to solve the problem.
- Review your logic as you go. It doesn't make sense to refine something that doesn't work to begin with.
- Once you have your algorithm done, you can rewrite that as comments. Use these comments at the beginning of the blocks where you write your code. This way your design is with your code that you (and others) can refer to.

5.3 Functions and Decomposition

Where do functions fit in all this? Sometimes you'll get a design (from a teacher or co-worker) asking for a specific function in your program. Other times, you'll notice that a piece of logic is being repeated in several places; and you know that writing functions allows us to reuse that logic whenever needed. At other times, you'll see that a task has been broken down into several subtasks. To organize this well you may want to put each of those subtasks into a function.