CS121 Discussion 3- Using Objects

Objectives:

1. Understand the parts of the Java API documentation.
2. Know how to read the API of an Object.
3. Create a Javadoc document.
4. Work with the java.util.Scanner class.
5. Eliminate redundant code by refactoring.
6. Develop new code incrementally.
7. Create a new class to encapsulate a specific functionality.

Introduction:

In this session, you will gain further experience working with the Object model. An Object is an instance of a class definition. It contains data and functionality. Often, you will use an existing class or library of classes in your program. After all, it makes sense to re-use existing code instead of writing it from scratch. You need to know how to work with existing code in order to use it. What you need to know is: how do I make a new instance of a class, and what public functions does the class provide? The set of public methods in a class definition are referred to as that class's interface, or its Application Programming Interface, or API.

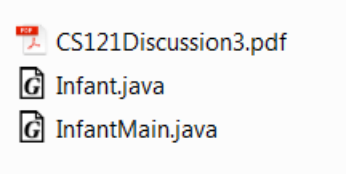
You will work with Java library classes as well as some other existing classes in this example.

Assume: You have created a directory (folder) on your machine named "CS121Projects" where you place all of your CS121 work for this semester. This should be located in your "Documents" directory.

Begin:

Create a directory called "discussion3" in the "CS121Projects" directory. Download the discussion3.zip file from the Moodle link "Discussion 3: Working with Objects" under week 3. Unzip the file and place the contents in the "discussion3" directory. Make sure you extract the files directly to the "discussion3" directory. You should now see the following files in the discussion3 directory, on this path:

/CS121Projects/discussion3



Task: Now create a project in jGrasp called “discussion3. Create the project in the “discussion3” directory, which is same as the project name. You do not want to create a new directory when you create the new project (since you already did that), so uncheck the “Create New Directory” option. Click on the Infant class to open it in the editor pane.

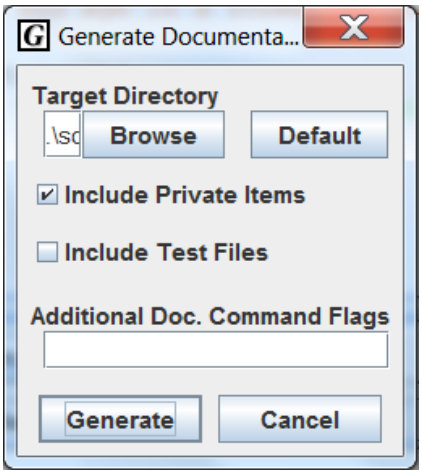
Javadoc

There is a special format that is used to document Java code called “Javadoc”. When a java source code file is commented in a specific way, Java can produce a set of hyperlinked documents (html web pages really). Each document corresponds to each class definition (and therefore each java source code file). The Javadoc file for a class has three major parts: Field Summary- the member attributes, Constructor Summary- how to make instances of the class, and Method Summary- the set of public methods that defines the class’s functionality, or how it can be used.

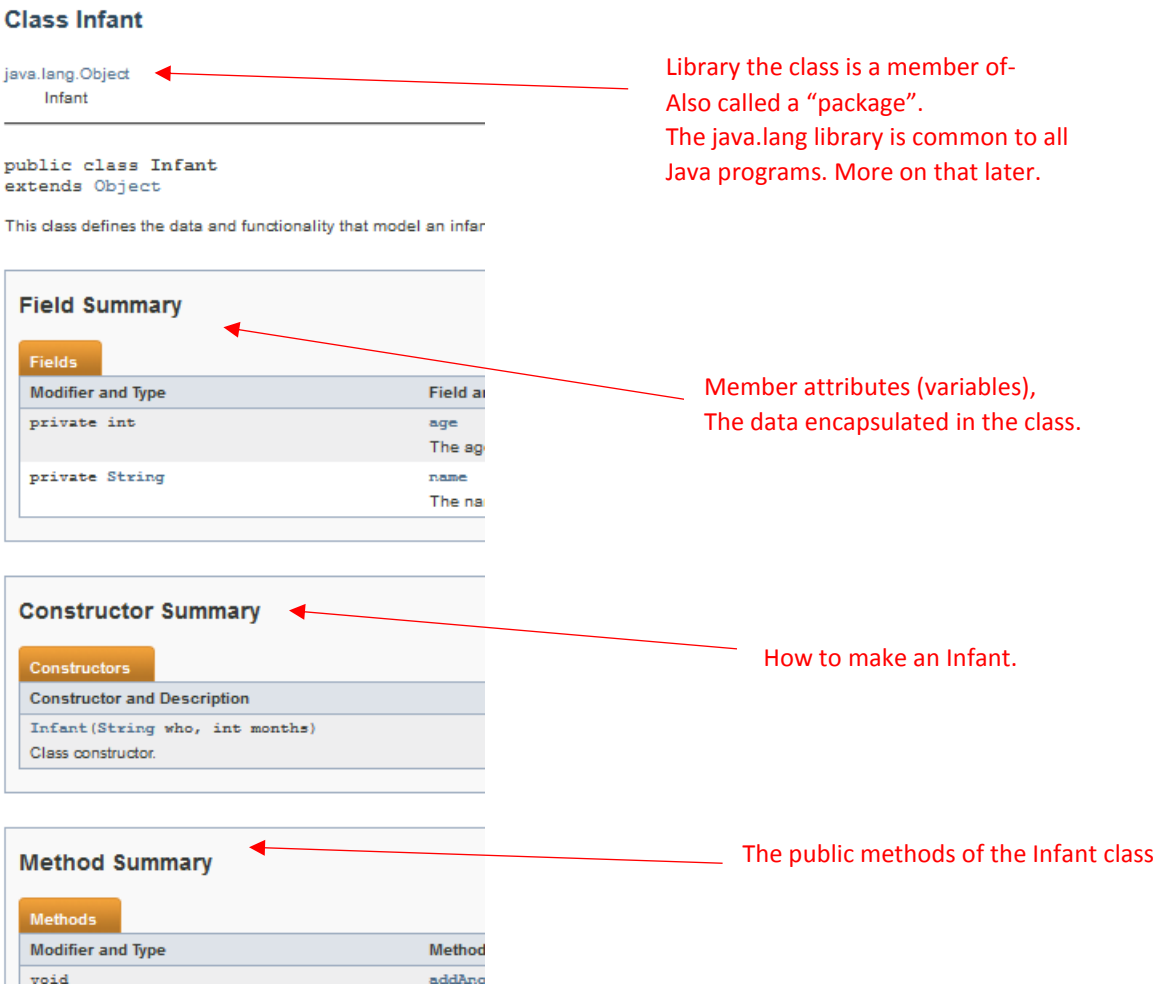
Javadoc is a standardized way that Java code is documented. This makes it easy for you, the developer, to be able to understand how to use existing code in your program. For example, let’s say we want to use the Infant class in a program. (yes, we are very familiar with this class by now). The Infant class that is included with your discussion3 code has been commented so that it will produce a Javadoc file.

Task: Now that you have the discussion3 project open, click on the book icon on the project pane toolbar to create Javadoc:

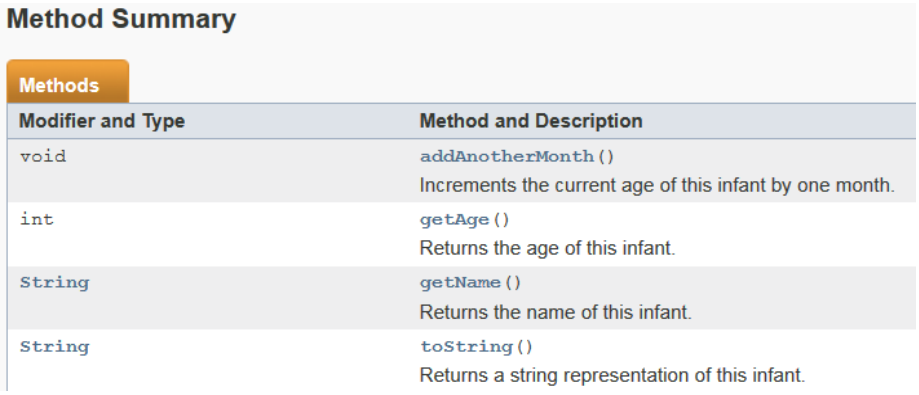
You’ll see a dialog open. Click “Generate”.



This will run the “Javadoc” process that will produce the html style documentation. The html file for Infant should open in your browser. If it doesn’t, look for the “discussion3_doc” directory, open it, and open the Infant.html file.



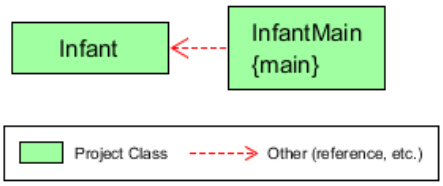
The Method Summary is especially important as it provides information of how the class can be used. This is the *interface* to the class.



The Infant class is not very complicated, and we really don't need Javadoc to understand how to use this class, however; it serves as a simple example of what Javadoc is and to illustrate the important parts of a Javadoc document. We will not be concerned with making Javadoc from our code in this class. However, it is important that you know about Javadoc and how to read it so that you can find out how to use code from existing libraries- mostly the Java API. You will want to consult the Java API- the Javadoc documentation for the major Java libraries that come with the JDK for classes like String, Scanner, and Math, among many others.

Working with Scanner and Object-Oriented design.

Now we'll work with the Infant and InfantMain code. The UML diagram of this code (generated by jGrasp) follows:



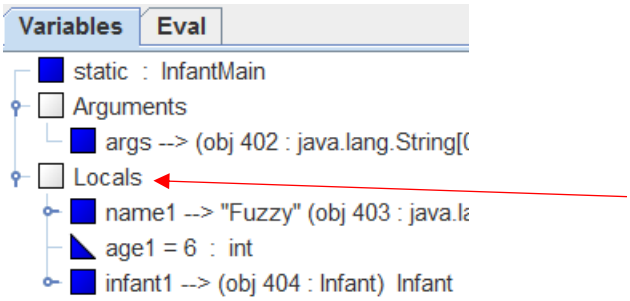
Recall that UML stands for Universal Modeling Language, and is a widely used diagramming technique for software development. Notice in the diagram that the two classes are represented in green boxes, with the main method shown in the InfantMain class. The red dotted arrow indicates that there is a relationship between the two classes. The arrow means that the InfantApp class holds a *reference* to the Infant class.

The code in InfantMain declares two variable of type Infant on line 7. Note that these are not initialized to a value. We don't have to assign them to Infant objects until later on.

```
5  public class InfantMain {
6     public static void main(String[] args) {
7        Infant infant1, infant2;
8        String name1 = "Fuzzy";
9        int age1 = 6;
10       infant1 = new Infant(name1,age1);
11       System.out.println(infant1.toString());
12    }
13 }
```

Then, two "local" variables name1 and age1 are declared and initialized. These are local because they exist in the InfantMain class. In contrast, the variables for name and age in the Infant class are not local. Local means "in the scope". Scope is defined by a pair of opening and closing curly braces {}. All of the code in InfantMain is in the scope of the main method, so the

variables infant1, infant2, name1 and age1 are local to the scope of main. You can see that jGrasp refers to these variables as “Locals” in the debugger:



Back to the code, line 10 is where the infant1 variable gets assigned to a new Infant object. The infant2 variable is left unassigned. It actually has a default value of *null*. A default value is a value that the system uses to assign to a variable when the programmer hasn’t specified a value. Null is not a value but a placeholder. More on that later. Finally, the statement on line 11 calls the toString method on infant1 and prints that string to the console, and we see:

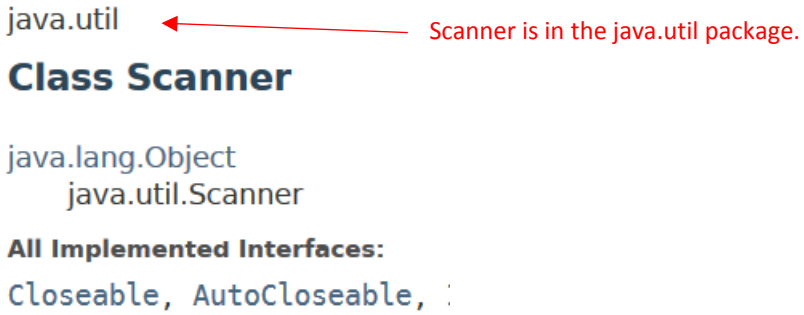
```
Infant name: Fuzzy, age (in months): 6
```

The name and age for this Infant was “hard-coded” into the program. Next, we’ll work with getting values in a more dynamic way.

Using a Scanner

Now modify the code above so that the program obtains the name and age for two infant objects from a user. (Think of a day care employee doing some data entry). A Scanner object is designed to provide the functionality required to get data entered from the keyboard. There is a lot of complex stuff that goes on to make that happen, including talking with the computer’s operating system. Fortunately, we can use the Scanner’s public methods (its API) to accomplish what we want.

To add a Scanner to our code, we must import it as it is in a different library, or *package*. This can be seen from the Scanner’s Javadoc page in the Java API:



Any Java class that isn't in the java.lang package must be imported. The String class, for example, is in the java.lang package, so we don't have to import it. (If we were to build our code in a package we define, then we do not have to import those classes).

Task: So, in order to use a Scanner, we have to include the proper import statement at the top of our InfantMain code.

Task: The next step is to make a new Scanner object, and then we can use the variable to call Scanner public methods to do our bidding.

```
1  /*
2  * This class creates and works with infant objects.
3  *
4  */
5  import java.util.Scanner;
6
7  public class InfantMain {
8     public static void main(String[] args) {
9        Scanner scan = new Scanner(System.in);
10       Infant infant1, infant2;
```

Now we are ready to use the Scanner object "scan" created on line 9. We want to first prompt the user to enter a name, then use scan to get the name the user enters. Then, we prompt the user to enter an age for that infant and use scan again to get the age typed in. How do we know what methods to call? They are in the Scanner Javadoc page under the public methods section.

String	**next()** Finds and returns the next complete token from this scanner.

The "next" method returns a *token*. (notice the data type String on the left). A line of text is made up of one or more tokens. Token is a term for a piece of data in a sequence. In this case, tokens are the words in a line of text. So, if we expect a single word, or token, to be typed in, we can use the next method. If more than one word is typed in, the next method returns only the first word. The rest of the words stay in the Scanner's *buffer*. A buffer is a storage area that saves data received from a stream for later use- like the buffer in streaming video. The stream in this case, is the stream of characters typed from the keyboard.

Task: To see this in action, execute two calls to "next" when more than one word was typed in:

```
9        Scanner scan = new Scanner(System.in);
10       Infant infant1, infant2;
11       System.out.println("Enter an infant name:");
12       System.out.println(scan.next());
13       System.out.println(scan.next());
```

Produces this output:

```
Enter an infant name:
Green Bay packers
Green
Bay
```

Now, what about getting the age? The Scanner public method nextInt should do the job:

int	**nextInt()** Scans the next token of the input as an int.

Again, notice the data type it returns on the left- and integer, or int. The method pulls in the next token as a String, but converts it to an integer for us.

Question: What Scanner method would we use if we want a complete line of text?

Task: Now that you know what Scanner methods to use, add the statements that call them to the InfantMain code. Note: assign the values returned by the scanner calls to the variables name1 and age1. This is an example of how the code should now function when you compile and execute:

```
----jGRASP exec: java InfantMain
Enter an infant name:
Thurston
Enter an infant age:
8
Infant name: Thurston, age (in months): 8
```

Make sure you get this working before moving on.

Add another Infant

Task: Now add to the code so that the user can enter a second infant. Create a new Infant object using the second name and age values the user types in (don't forget to prompt the user as well). Use the variable "infant2" to point to, or reference, the new Infant. Declare extra local variables name2 and age2. Print out the second infant's data as you do for the first infant. Your code should do something like this:

```
Enter an infant name:
Bart
Enter an infant age:
4
Enter an infant name:
Lisa
Enter an infant age:
5
Infant name: Bart, age (in months): 4
Infant name: Lisa, age (in months): 5
```

Refactoring the code.

You have accomplished the modifications to the original code using the same two classes you started with: Infant and InfantMain. Let's look at the low level tasks your code does:

1- create a Scanner
2- prompt user for name1
3- assign name1 to variable
4- prompt user for age1
5- assign age1 to variable
6- create infant1
7- prompt user for name2
8- assign name2 to variable
9- prompt user for age2
10- assign age2 to variable
11- create infant2
12- print infant1
13- print infant2

All of these tasks are done in the InfantMain class. It is doing a fair amount of work and using a lot of variables. We have to name these variables 1 and 2 to distinguish them from each other. Another aspect of the tasks listed above is that we are repeating many of the same steps twice. Whenever you find yourself doing the same thing more than once, you want to think about how you can reorganize your code to "encapsulate" the repeated steps in a class. (Note: A technical term for code re-organization is "refactoring").

Code duplication creates a greater possibility of errors- what if you change one part but not the other- and is more difficult to test and modify in the future. Imagine if our program needed to handle many infants? We'd be repeating the same steps many more times.

We will use an Object-Oriented approach to re-organize our code.

The way to do this is to create a class that does some or all of the repeated steps in our task list. For example, these tasks for creating infant1 are repeated for infant2:

2- prompt user for name1
3- assign name1 to variable
4- prompt user for age1
5- assign age1 to variable
6- create infant1

The new class will carry out these tasks and return a new Infant object to the calling code- in this case the InfantMain class.

Task: Create a new Java file and write the class definition. Let's call the class InfantGenerator, since it will generate Infants from user input.

Next, we have to define the API of the InfantGenerator class, which tells us how it can be used. This forces us to make some decisions. In InfantMain, we make a Scanner to get user input. Do we pass this Scanner in to the InfantGenerator or does InfantGenerator create and use it's own Scanner?

There are several pros and cons to consider. The InfantMain code would be greatly simplified if it didn't have to make a Scanner. On the other hand, there will be some resource overhead if each InfantGenerator has to create a Scanner.

Let's say that we decide to put the Scanner in the InfantGenerator class. That creates a better separation between InfantMain and InfantGenerator. That is a goal of Object-Oriented design: to avoid dependencies when possible. If InfantMain makes a Scanner and passes it to InfantGenerator, the two classes are more dependent on each other.

Now, back to the API for InfantGenerator. Once an InfantGenerator is created, we need a method to tell it to go ahead and do its work, and a method to return a new Infant object. Let's call these methods getUserInput and getInfant.

Task: Write the "skeleton" first. This means write the very minimum to get the new class to compile. This way of developing code is called “incremental” development. Don’t write a large amount of code all at once. Write the skeleton, then add little bits, compiling and testing each time before moving on.

The skeleton for Infant Generator could be:

```
1  import java.util.Scanner;
2
3  public class InfantGenerator{
4
5    Infant infant;
6    Scanner scan;
7
8    public InfantGenerator(){
9       scan = new Scanner(System.in);
10   }
11
12   public void getUserInput(){
13   }
14
15   public Infant getInfant(){
16     return infant;
17   }
18 }
```

Member variables: Infant and Scanner

Use the constructor to initialize the scanner object

Code to prompt user, get data, create Infant here.

The getUserInput method would carry out the tasks 2-6 in the task list above, and also tasks 7-11. In fact, InfantGenerator will work for each infant you need to make in this way. You can copy and paste code from InfantMain into the getUserInput method. You'll have to make some modifications to adapt it to its new location.

Task: Then, clean out the code in InfantMain. You don't need the Scanner as it's in the InfantGenerator now. Here is the InfantMain code that uses the InfantGenerator class to create one infant:

```
5  public class InfantMain {
6     public static void main(String[] args) {
7       Infant infant1, infant2;
8       InfantGenerator infGen = new InfantGenerator();
9       infGen.getUserInput();
10      infant1 = infGen.getInfant();
11      System.out.println(infant1.toString());
12    }
13 }
```

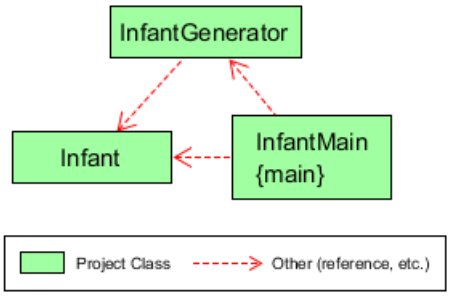
Task: Compile and test the one infant code above. Notice how the InfantGenerator public methods are used by the InfantMain class. Now add statements to InfantMain that use the same InfantGenerator object to get the second infant in the same way. This is a sample of the input and output from the console:

```
Enter an infant name:
Homer
Enter an infant age:
9
Enter an infant name:
Marge
Enter an infant age:
11
Infant name: Homer, age (in months): 9
Infant name: Marge, age (in months): 11
```

This is the same *behavior* that we had before our use of InfantGenerator. We made a structural change to our code without affecting its overall functionality. That is the definition of *refactoring*.

You have now made a machine, the InfantGenerator, to crank out any number of Infant objects based on user input. The tasks 2-6 and 7-11 from the task list have been delegated to the InfantGenerator class. Because of this refactoring, the code that accomplishes these tasks is not repeated in more than one place.

The project code now has three classes as you can see in the UML diagram:



Next Task: Trolls!

The Troll class defines the attributes and (some) behaviors of a Troll. Your task is to create a TrollMain class that creates two Trolls with the following attributes:
"Hakkar", 220, "MACE"
"Loki", 200, "BLUDGEON"
Then prints their values to the console. Next, allow for two more Trolls to be created from user typed values and print them out. Define a TrollGenerator class to get the user-typed values and create the new Troll objects that are returned and printed in the TrollMain class.