CS121 Discussion 2- The Object Model

Objectives:

1. Understand the purpose and structure of Objects.
2. Understand that an Object is an *instance* of a Class.
3. Know how the constructor method is used to create new Objects.
4. Understand that access to Object data is controlled by its public methods.
5. Understand how to access and modify data in an Object.
6. Investigate how to use the debugger to view values inside of an Object.
7. Write code to create a new Object and print out data from the created Object.
8. Write a new class definition based on how it is used by another class.

Introduction:

In this session, you will gain experience working with the “Object” model. This is the model of code organization that is used by all Object-Oriented programming languages. The purpose of Objects are to *encapsulate* data and functionality that is closely related. Think of an Object as a container that allows the program designer to control access to its contents. For example, a Calculator class would have data and functions that perform arithmetic calculations.

Some technical terminology: To create an Object in code you first have to write a Class definition. This is like a template from which many *instances* of that Class can be made when the code runs. Each instance of a Class is called an Object.

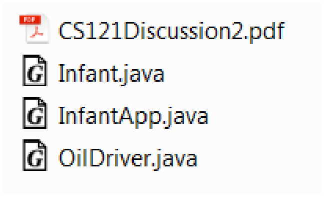
Each Class definition contains data and functions. The data is declared private, or hidden, and some of the functions are declared public, visible to all. The set of public methods of a Class are called its *interface*. The interface determines how other code Objects can interact with that Class.

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:

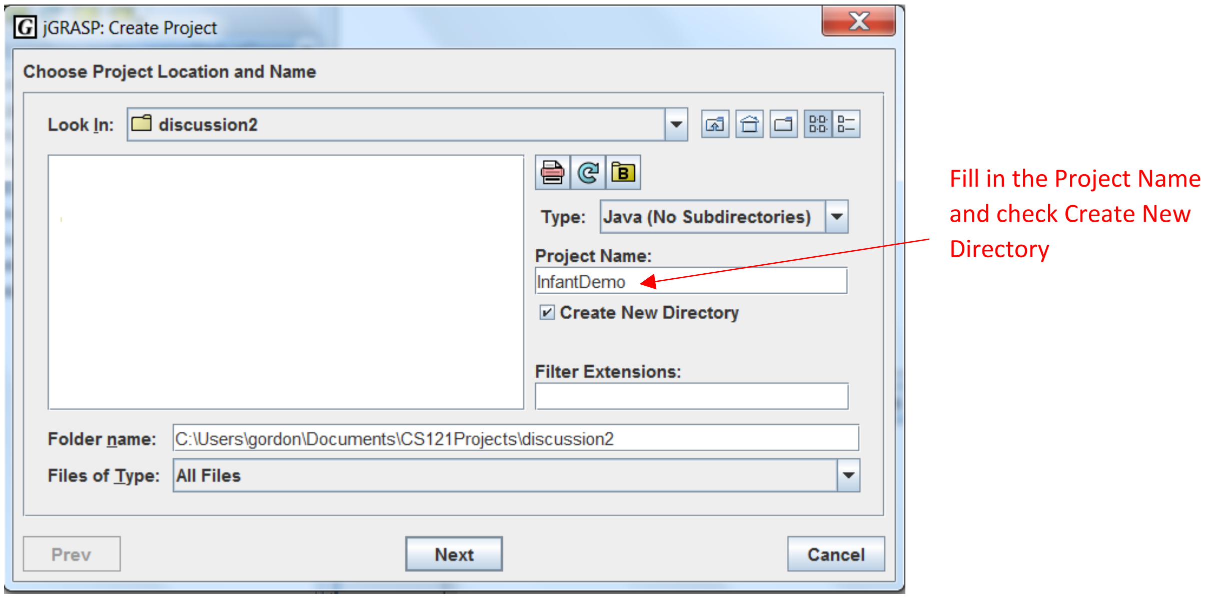
Create another directory called “discussion2”in the “CS121Projects” directory. Download the discussion2.zip file from the Moodle link “Discussion 2: The Object Model” under week 2. Unzip the file and place the contents in the “discussion2” directory. Make sure you extract the files directly to the “discussion2” directory. You should now see the following files in the discussion2 directory, on this path:

/CS121Projects/discussion2



CS121Discussion2.pdf
Infant.java
InfantApp.java
OilDriver.java

Now create a project in jGrasp called InfantDemo. Create the project in a directory named InfantDemo, which is same as the project name. jGrasp has a checkbox to do this when you create a new project in the Create Project dialog box that appears when you select Project/New:



jGRASP: Create Project

Choose Project Location and Name

Look In: discussion2

Type: Java (No Subdirectories)

Project Name: InfantDemo

[✓] Create New Directory

Filter Extensions:

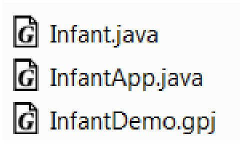
Folder name: C:\Users\gordon\Documents\CS121Projects\discussion2

Files of Type: All Files

Then create the project, select Add Files to Project Now, navigate to and select the two files Infant.java and InfantApp.java. A box will pop up with options. Select Move and then click Done. This will move the two Java files into the newly created InfantDemo directory. Now you have created a project called InfantDemo. This is the path you should be working with for the project:

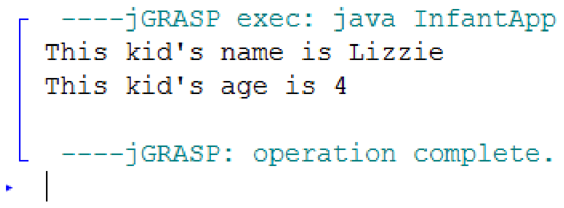
/CS121Projects/discussion2/InfantDemo

You should see these files in the InfantDemo directory:



Infant.java
InfantApp.java
InfantDemo.gpj

Open the Infant.java and InfantApp.java files in the editing pane by double-clicking on the files in the project pane. If line numbers do not appear, select View/Line Numbers. Now compile and run the program. You will see this output in the interaction pane:

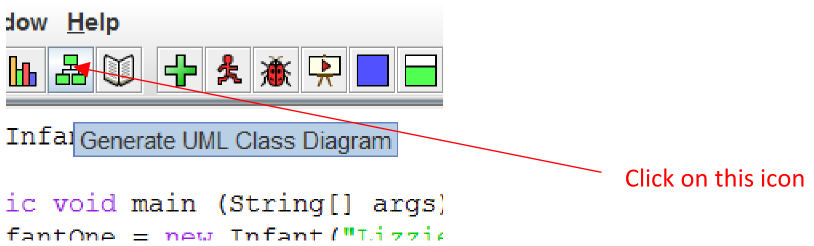


```
 ----jGRASP exec: java InfantApp
This kid's name is Lizzie
This kid's age is 4

 ----jGRASP: operation complete.
```

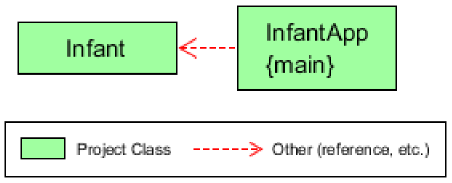
How new Objects are created: the constructor method

We have a two class program. One class, InfantApp, has a main method and creates an instance of the Infant class. To get a diagrammatic view of the two classes, click on the Generate UML Diagram button at the top of the editor pane:



```
Infa
ic void main (String[] args)
fantOne = new Infant("Lizzie
```

You will see a new dialog open with this diagram:

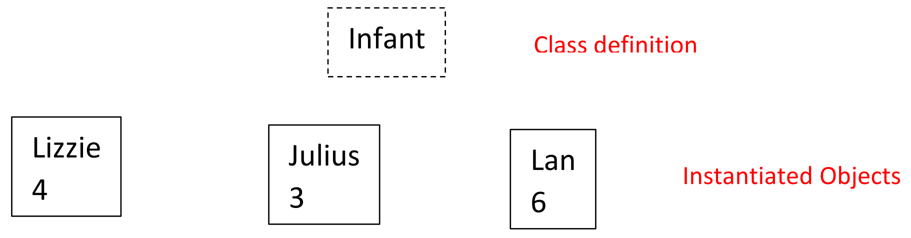


UML stands for Universal Modeling Language, and is a widely used diagramming technique for software development. Notice in the diagram that the two classes are represented in green boxes, with the main method shown in the InfantApp class. The red dotted arrow indicates that there is a relationship between the two classes. The arrow means that the InfantApp class holds a *reference* to the Infant class. (You can dismiss the UML Diagram box). This reference is created on line 4 of the infantApp class:

```
 Infant infantOne = new Infant("Lizzie",4);
```

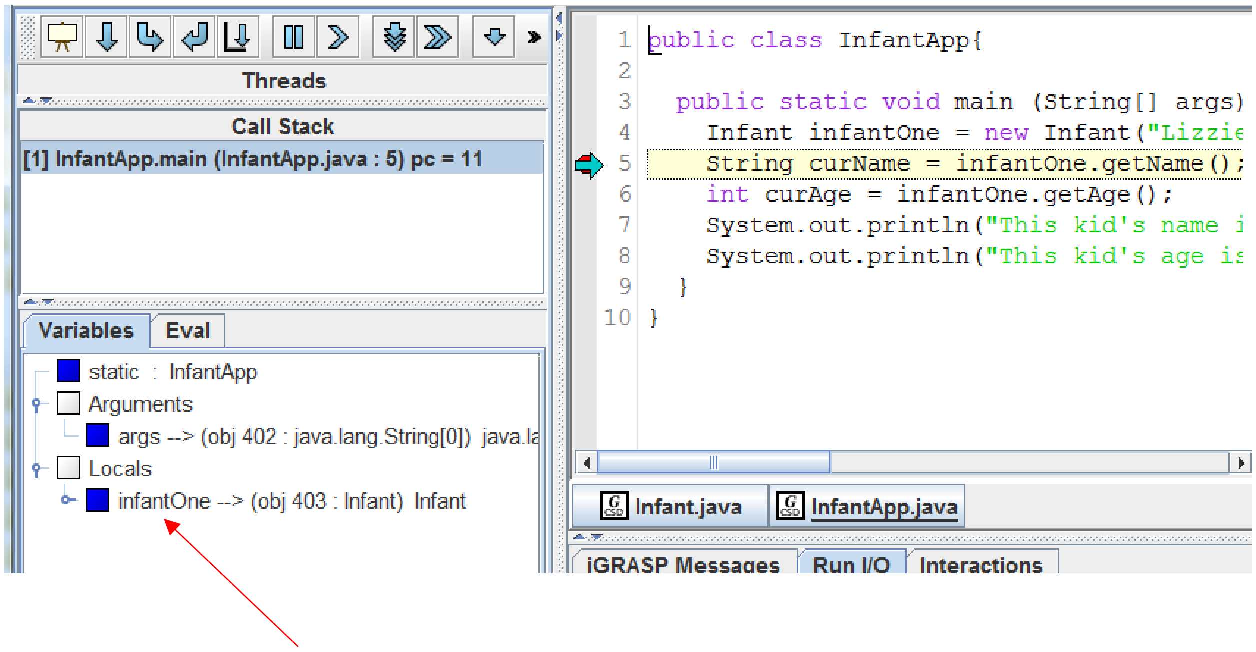
This statement declares a variable of data type Infant called infantOne and assigns to it the value of a brand new instance of the Infant class having the data "Lizzie" and 4, a String and an integer respectively. Notice the “new” keyword. This is an operator that asks the operating system to allocate new memory for an instance of the Infant class. Once this statement completes, the program has created an *instance* of the Infant class containing the name “Lizzie” and the age 4. This is a new Object, and is referenced by the variable infantOne.

In fact, the right hand side of the statement on line 4 *calls* the Infant class *constructor*, the definition of which you can see on lines 6-9 in the Infant class. A constructor is a special function that allows new instances of a class to be created in memory. It also allows data values to be passed in and stored in the *member variables* of a class. The declarations of the member variables in the Infant class are on lines 3 and 4. These variables get assigned to specific values in the constructor, on lines 7 and 8. It is possible for a program to make any number of instances of a class. Each instance of a class is called an Object. Each Object has its own specific data. For example, the Infant class could be instantiated three times:



While all three objects in this example have different data, they all share the same functions. The functions, getName, getAge, addAnotherMonth are defined in the Infant class on lines 11, 15, 19. They allow code that uses instances of the Infant class to access and modify the data that is contained inside of the Objects.

Now, look at how the InfantApp class uses the Infant class constructor. On line 4, an instance of the Infant class called infantOne is created. This is a variable that references an Infant Object with the data: “Lizzie” and 4. Set a breakpoint at line 5- right after the constructor is called. Run the debugger- the execution will stop at line 5. Look in the Variables pane to the left. You will see “infantOne” under locals. (What does “Locals” mean?). Notice the information to the right of “infantOne”. It looks something like this:



```
[1] InfantApp.main (InfantApp.java : 5) pc = 11
```

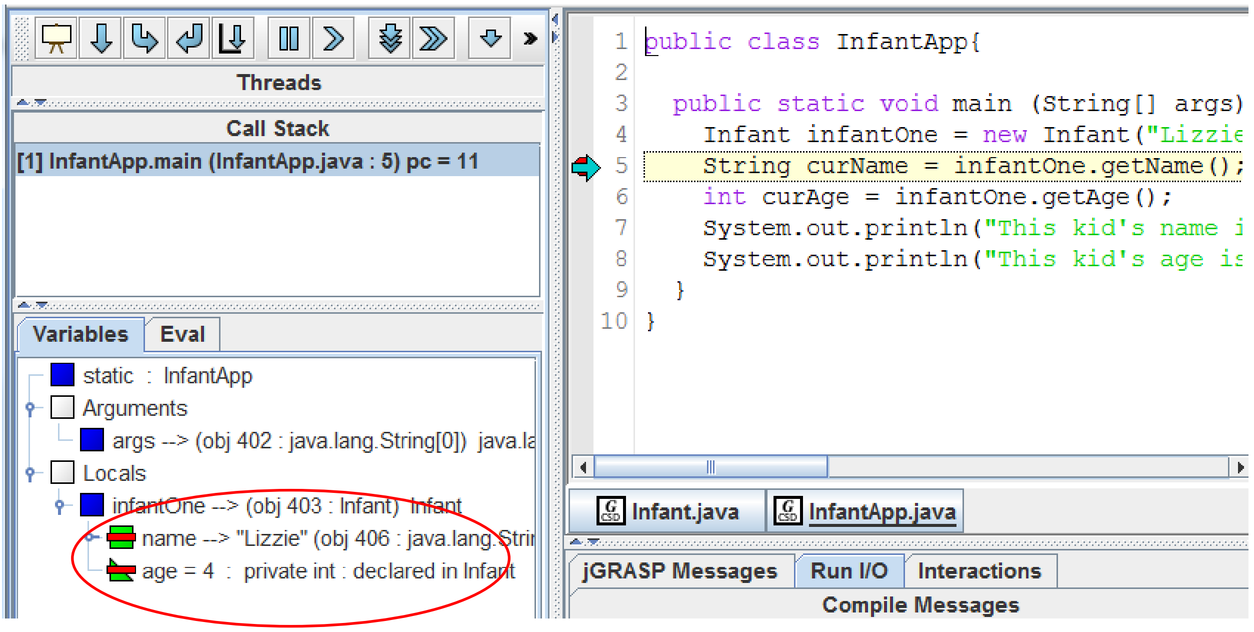
```
static : InfantApp
Arguments
  args --> (obj 402 : java.lang.String[0])  java.la
Locals
  infantOne --> (obj 403 : Infant)  Infant
```

```
public class InfantApp{

  public static void main (String[] args)
    Infant infantOne = new Infant("Lizzie
    String curName = infantOne.getName();
    int curAge = infantOne.getAge();
    System.out.println("This kid's name i
    System.out.println("This kid's age is
  }
}
```

This tells you that the variable infantOne is an Object (not a primitive type like an int or double), and that it is of data type Infant- an instance of the Infant class- made from the Infant definition. The number, 403 in the example, is the unique identifier for this particular Object and will likely be different on your machine. When an Object is created (with the memory allocation operator “new”), it gets stored in main memory. Think of the number that appears after “obj” as the memory location for this particular Infant Object.

Click on the little circle to the left of infantOne entry in the variables pane to expand it. You will see two entries under infantOne: “name” and “age”. These are the private data members of this Infant Object. You can see their current values. Notice their values as well as their data types are shown. What data type is “name”? Is it a primitive data type?



Threads

Call Stack

[1] InfantApp.main (InfantApp.java : 5) pc = 11

Variables | Eval

- static : InfantApp
- Arguments
  - args --> (obj 402 : java.lang.String[0]) java.la
- Locals
  - infantOne --> (obj 403 : Infant) Infant
    - name --> "Lizzie" (obj 406 : java.lang.Stri
    - age = 4 : private int : declared in Infant

```
 1  public class InfantApp{
 2
 3    public static void main (String[] args)
 4      Infant infantOne = new Infant("Lizzie
 5      String curName = infantOne.getName();
 6      int curAge = infantOne.getAge();
 7      System.out.println("This kid's name i
 8      System.out.println("This kid's age is
 9    }
10  }
```

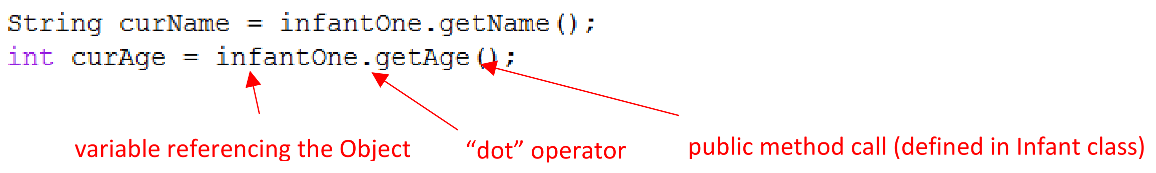
Infant.java | InfantApp.java

jGRASP Messages | Run I/O | Interactions

Compile Messages

Using public methods to access and modify private data in an Object

Now click on the “Step” icon on the upper left (the blue downward pointing arrow). The code on line 5 declares a String variable curName and assigns to it the current value of the name *inside* the infantOne Object. Notice the syntax of these two assignment statements- especially the right hand side:



```
String curName = infantOne.getName();
int curAge = infantOne.getAge();
```

variable referencing the Object
“dot” operator
public method call (defined in Infant class)

This is how you can “use” a reference to an Object to access the private data inside it. Note that if there is no public method to access a private data member, then you cannot get access to it.

With the debugger, we the developer can see inside the infantOne Object, but the code in InfantApp cannot because the data are *private*. The only way that InfantApp (or any other code that interacts with an Infant Object) can see these data is by calling a public method that is defined in the Infant class. (A class may define private methods, in which case they are not visible outside of the Object). The set of public methods defined in a class is called its *interface*. Another term that can be used is “API” (Application Programming Interface). The API controls how other code can interact with the data and functionality of a class. In line 5, we see that the InfantApp code is using a public *get* method, called “getName” to access the private data member “name” from the infantOne Object. The method is called by using the variable name, dot operator, and method name. It returns the String value “Lizzie”.

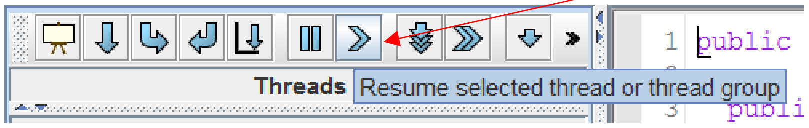
Note: the values for name and age were initialized for the infantOne Object via the constructor call. The only way the name of an Infant Object can be initialized is at creation time. There would have to be a public method to allow for the name to be changed after the Object was created. Methods that allow data members inside objects to be *read* or *accessed* are called ***accessor*** methods, or “get” methods as their names usually begin with “get”, to make their purpose clear. This is not a requirement but it is a convention among developers.

Why do we have an API? One word for this is called ***data hiding***: access to data should be controlled. An API allows the designer and developer to specify exactly which data can be accessed and how it can be initialized and modified. In other scenarios, data is not *hidden* in Objects, but *globally* accessible. In this case, any code can modify the data. This often leads to unanticipated effects because one area of code may not know that another area of code has changed a value. This is especially true for large code bases where different code modules were developed by different developers. The Object Oriented approach was introduced to deal with such unanticipated *side effects*.

Now, click on the step arrow again. You see that the InfantApp variables curName and curAge have been assigned to the values “Lizzie” and 4 respectively.

The next line will use the curName variable to print to the console. Step to line 8.

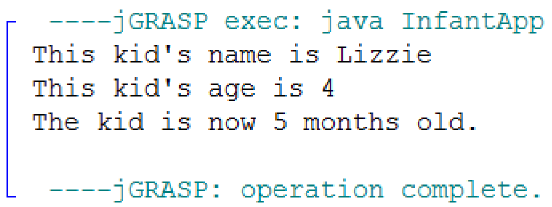
Line 8 prints the age to the console by calling the method getAge. This is to demonstrate that you can use a call to the infantOne method to get the value or you can use a local variable that is holding the value, “curAge” in this case. Now click on the single chevron (“Resume”) to let the code run to the end.



You will now write code that modifies the values inside of an Object. The only way the InfantApp code can modify values in the infantOne Object is by calling a *public* method in the Infant API. We want to change the age of infantOne from 4 to 5 months. Looking at the Infant class, we see there is a method “addAnotherMonth”, which will do just as it says- add one to the current age. Methods that allow you to change values inside of an Object are called ***mutator*** methods. Write a statement in the InfantApp class (after line 8) that will add another month to Lizzie, and then write the statement that will print out Lizzie’s current age in this manner:

This kid is now 5 months old.

When you compile and run the program, you should see this output:



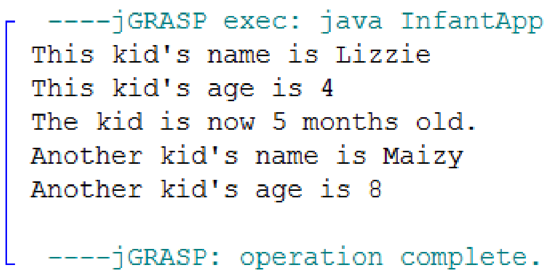
```
 ----jGRASP exec: java InfantApp
This kid's name is Lizzie
This kid's age is 4
The kid is now 5 months old.

 ----jGRASP: operation complete.
```

Creating other Infant Objects

Next, after line 10 in InfantApp, write statements that reassign the variables curName and curAge to have the following values: “Maizy”, and 8, respectively. Create a new Infant Object using these data and assign it to a variable called “infantTwo”.

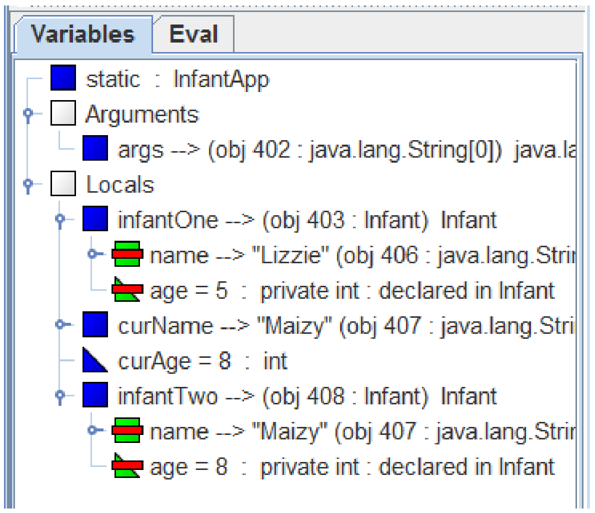
Next add two statements that print the name and age of infantTwo repectively. Use ***only*** the infantTwo variable to reference the appropriate Infant class methods in your statements. Your program should produce the following output when run:



```
 ----jGRASP exec: java InfantApp
This kid's name is Lizzie
This kid's age is 4
The kid is now 5 months old.
Another kid's name is Maizy
Another kid's age is 8

 ----jGRASP: operation complete.
```

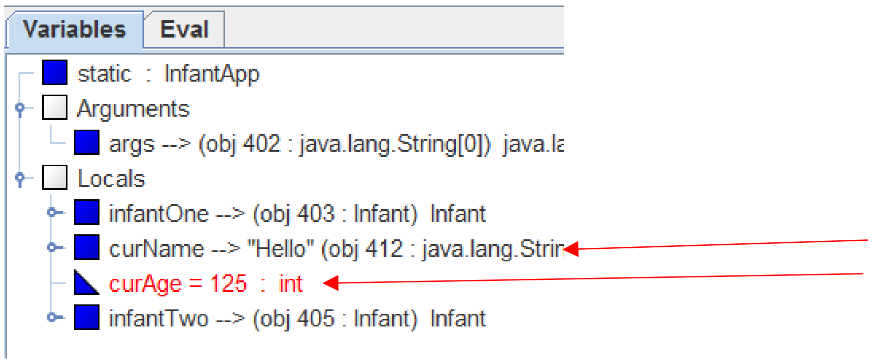
Put a breakpoint on the last line of your program and run in debug mode. When the execution halts, look at the two Infant references in the Variables pane. Expand these entries to see the data that is stored in each Object.



```
Variables  Eval
static : InfantApp
Arguments
  args --> (obj 402 : java.lang.String[0])  java.la
Locals
  infantOne --> (obj 403 : Infant)  Infant
    name --> "Lizzie" (obj 406 : java.lang.Strin
    age = 5  :  private int : declared in Infant
  curName --> "Maizy" (obj 407 : java.lang.Stri
  curAge = 8  :  int
  infantTwo --> (obj 408 : Infant)  Infant
    name --> "Maizy" (obj 407 : java.lang.Strin
    age = 8  :  private int : declared in Infant
```

Notice that each Object has a different "id" number (403 and 408 in the example above) and that each has its own version of the member variables "name" and "age". Think of these two Objects as two containers that each contain their own data.

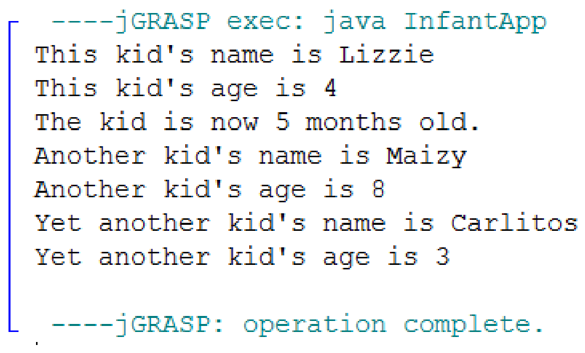
It is important to note that the variables "curName" and "curAge" in the InfantApp class are not related to the values stored in either Infant object. To see this, you will assign them to different values and then view them in the Variables pane. Stop the debugger (by clicking "End"). Reassign the values of curName and curAge to be "hello" and 125. Leave the breakpoint on the last print statement. Recompile and run again in debug mode. Now look at the Variables pane when execution stops. You see that curName and curAge have the values "Maizy" and 8. Click on the step control twice so that the two assignment statements are executed. You should see that the values for curName and curAge have changed in the Variables pane:



```
Variables  Eval
static : InfantApp
Arguments
  args --> (obj 402 : java.lang.String[0])  java.la
Locals
  infantOne --> (obj 403 : Infant)  Infant
  curName --> "Hello" (obj 412 : java.lang.Strin
  curAge = 125  :  int
  infantTwo --> (obj 405 : Infant)  Infant
```

Notice that the values in both Infant Objects have ***not*** changed.

Now create a third Object in the same way as for infantTwo. Reassign the curName and curAge variables to “Carlitos” and 3. Declare an Infant variable “infantThree” and assign it to a new Infant Object passing in the curName and curAge variables to the Infant constructor. Add two more print statements (use only the infantThree variable and the “get” methods) so that your program produces the following output:



```
 ----jGRASP exec: java InfantApp
This kid's name is Lizzie
This kid's age is 4
The kid is now 5 months old.
Another kid's name is Maizy
Another kid's age is 8
Yet another kid's name is Carlitos
Yet another kid's age is 3

 ----jGRASP: operation complete.
```

You now have a Java program that consists of two class definitions in two files: InfantApp.java and Infant.java. The InfantApp code creates three *instances* of the Infant class, each containing its own data values. Imagine a program that has to store and manage the data for fifty infants. You would have fifty instances of the Infant class in that program. This is the Object-Oriented approach to modeling a real-world situation where fifty infants are involved (such as a daycare center). Each infant is “modeled” by an Infant Object in the program.

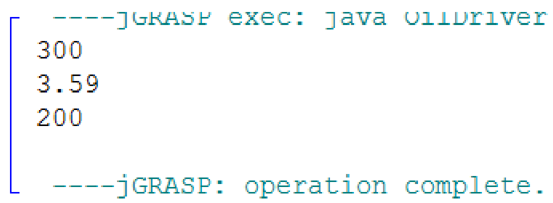
Part 2: Reverse engineering an Object

In software development, as well as in other forms of engineering, you may build a project from scratch- forward engineering- or figure out how it was built from an example- reverse engineering. In this part, you will create a class definition using an existing class as a guide.

The OilDriver class definition was supplied in the discussion2 distribution in the file OilDriver.java. Add it to your project now. It is supported by the OilTank class, which models a home heating oil tank, as indicated by the single line comment on line 4:

```
// owner - tank capacity - price per gallon - gallons in tank
```

Write the OilTank class definition using the code in the OilDriver class as a guide. The OilTank class has to have the data and methods that would make OilDriver work properly. Once you are able to compile and run the code, you should see this output:

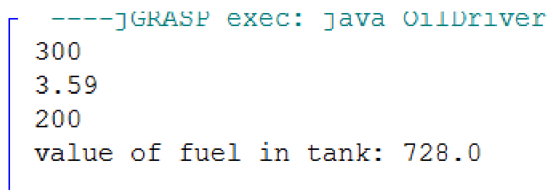


```
 ----jGRASP exec: java OilDriver
300
3.59
200

 ----jGRASP: operation complete.
```

(Since you have two classes with main methods, you will have to select OilDriver when you click on the run icon in the project pane).

Now that you've created the OilTank class, add a statement in OilDriver that prints to the console the value of the fuel in hankTank (the product of the current amount in the tank and the price of the oil). Use only the hankTank" reference and calls to OilTank methods to get the values for your calculation. This is the output your code should produce:



```
 ----jGRASP exec: java OilDriver
300
3.59
200
value of fuel in tank: 728.0
```

Part 3: Using methods to modify Object data

Now you will add to the set of public methods in the OilTank class and use the new method along with the other public methods defined in the OilTank class to access and modify the data in several OilTank Objects.

Add the method called "setAmtInTank" that allows the gallons in a tank to be modified. This method takes an integer parameter, call it "newAmt", and does not return a value. The method assigns the value of the parameter to the member variable "gallonsInTank".

Create two more OilTank objects each with the following attribute values:

owner: "Maya"	owner: "Yanjie"
tank capacity: 1300	tank capacity: 700
price per gallon: 3.44	price per gallon: 3.59
gallons in tank: 550	gallons in tank: 119

In a series of statements using *only* the public methods of the OilTank objects (and any local variables you want to declare), do the following:

1. Decrease the price per gallon of Yanjie's tank by 14 cents.
2. Add 20 more gallons to Maya's tank.
3. Swap the number of gallons in Hank's tank with Yanjie's tank.
4. Print out the total value of fuel in all three tanks (price times current gallons).
5. Print out the amount it would cost to completely fill each tank.

Extra: add methods that increase and decrease the amount of gallons in a tank by an amount passed in as a parameter. Add methods to return the value of the fuel in the tank as well as the total value of a full tank.

Add a method that prints the values in an OilTank object in a nicely formatted manner, such as:

```
owner: "Maya"
tank capacity: 1300
price per gallon: 3.44
gallons in tank:  550
```