

# CS121 Discussion 1- Getting started with Java and jGrasp.

## Objectives:

1. Compile and run a Java program given a source code file.
2. Create a new Java source code file, compile and run it.
3. Use the project manager in jGrasp to organize Java files as a project.
4. Know how to use the jGrasp debugger.
5. Understand basic error messages.
6. Declare and initialize variables.
7. Know basic Java data types.
8. Understand assignment statements.
9. Write arithmetic computations in Java.
10. Write statements that print data to the console.
11. Use good code writing practices.

## Introduction:

In this session, you will use jGrasp to create a project and add a Java source code file to your project. You will compile and run this code, and learn how to use the jGrasp debugger to control the execution of the code and to inspect the values of variables at each step of the program's execution. Then you will create a new Java source code file and add it to your project. You will also write basic Java assignments statements. Refer to chapter 1 in the "Interactive Java" text for background information (page 2.3 may also be helpful).

It is important that you learn how to organize your code as a project and how to use the debugger. These are essential skills for software developers. All Object-Oriented software consists of multiple code files as well as other resources. Projects organize your code so it can be compiled and run (and deployed) as a single code base. The debugger is an invaluable tool that allows you to control the execution of your program and to observe the values of variables in your program in real time.

**Assume:** You have installed the Java SDK and jGrasp on your machine. If not, find a partner who does. Make sure you install this software as soon as possible.

## Begin:

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

Create another directory called "discussion1" in the "CS121Projects" directory. Download the discussion1.zip file from the Moodle link "Discussion 1: Getting started with Java and jGrasp"

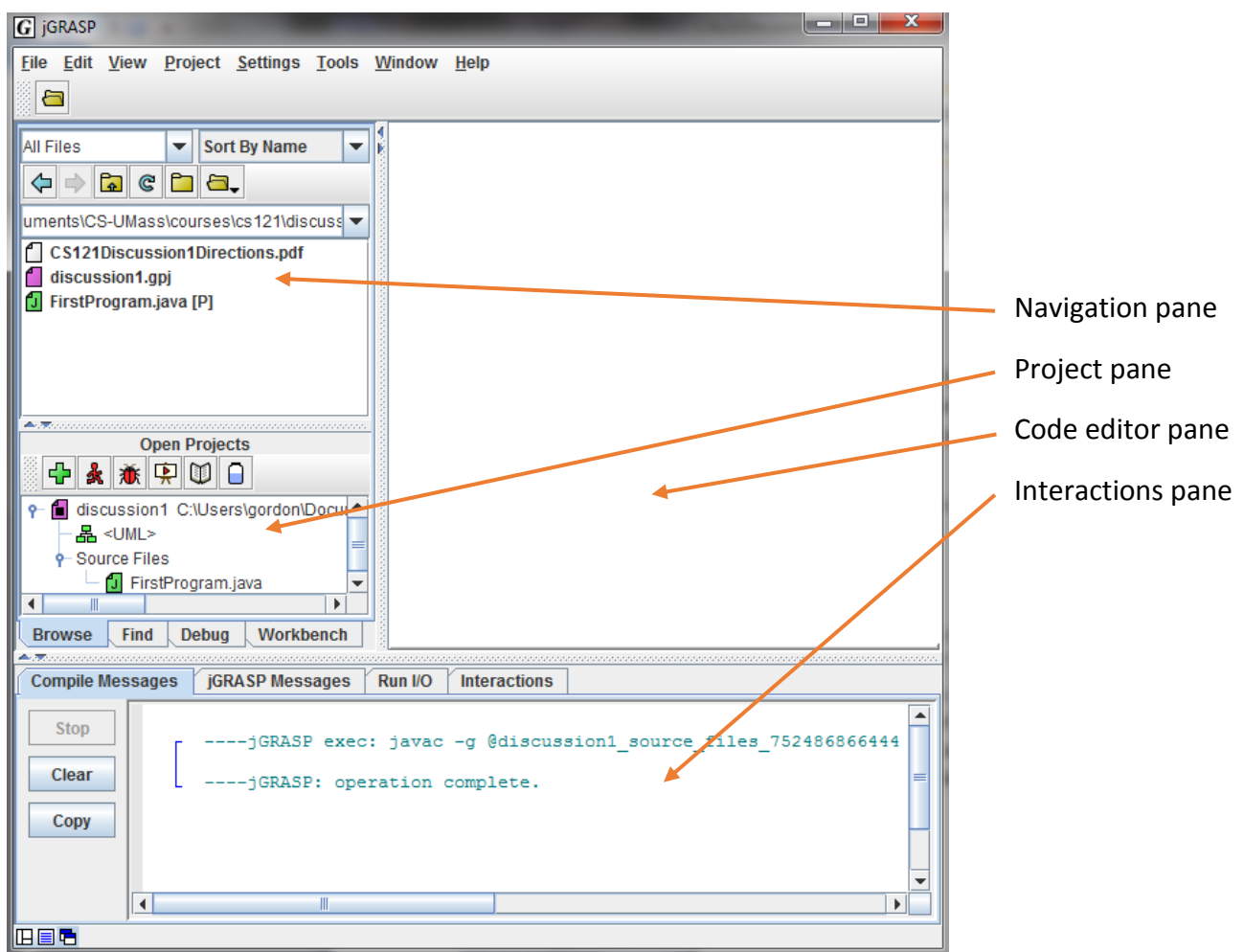
under week 1. Unzip the file and place the contents in the “discussion1” directory. You will see the following files: “CS121Discussion1.pdf” and “FirstProgram.java”.

## Part 1: Working with an existing file.

We’ll compile and run the program coded in the file “FirstProgram.java”. This file contains a Java program that prints a message to the “console”, or user’s screen. Since we are using jGrasp to run this program, the message will appear in the jGrasp interactions panel (more on that below).

Follow these steps to create a project, compile and run the FirstProgram code:

Run jGrasp. This is the jGrasp user interface:



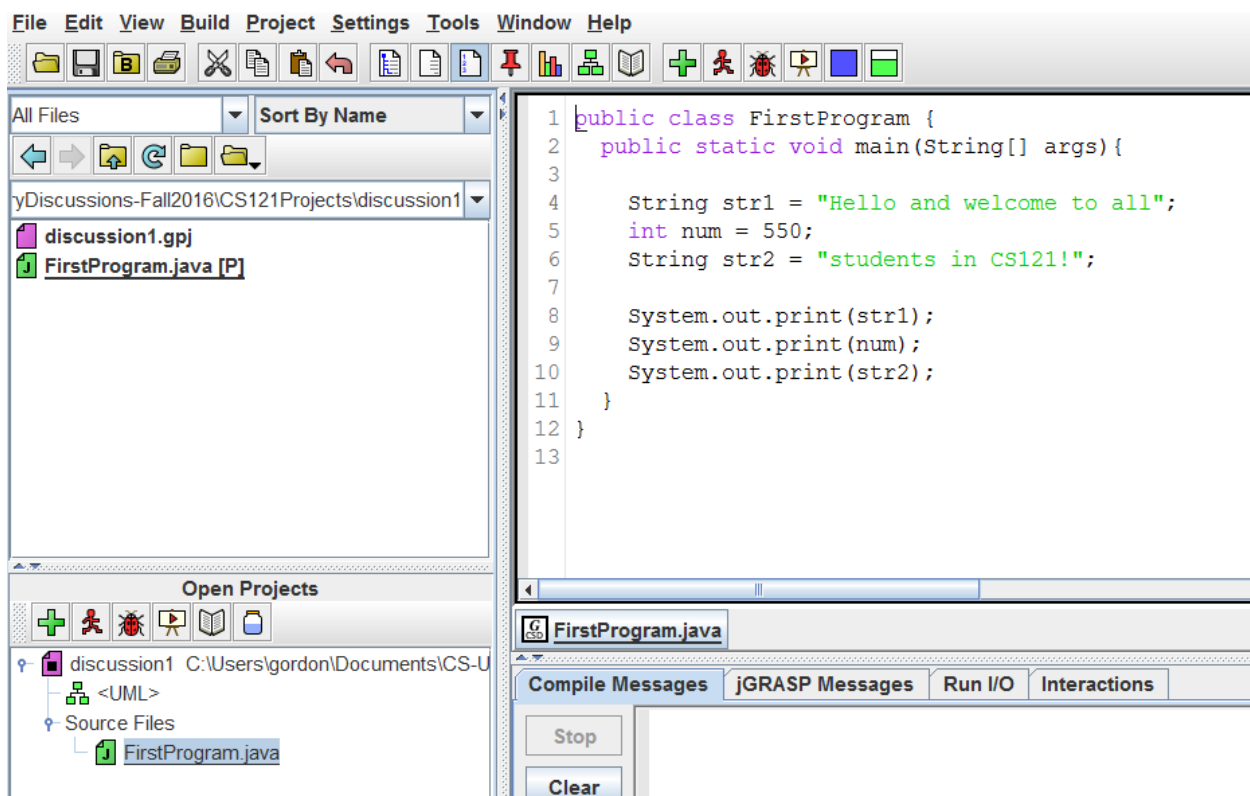
Use the navigation pane (upper left) to navigate to the discussion1 directory you created under the CS121Projects directory. Select the “Project” menu, “New” option. You see a dialog box appear. Type in “discussion1” as the Project Name, click “Next”. Make sure the “Add Files to Project Now” check box is selected and click “Create”. You’ll see a file browser. Navigate to the “discussion1” directory if you aren’t already there and select the “FirstProgram.java” file.

Click “Add”, then “Done”. You should see the project and the FirstProgram.java under the Source in the project pane in jGrasp (as shown above).

In addition to the “FirstProgram.java” file, you will see a file called “discussion1.gpj” in the navigation pane, above the project pane. This is a jGrasp project file that was created by jGrasp and contains information about the project. It is used only by jGrasp.

Now open the “FirstProgram.java” file in the jGrasp code editing pane by double-clicking on the “FirstProgram.java” file in the project pane.

You should see something like this in jGrasp:

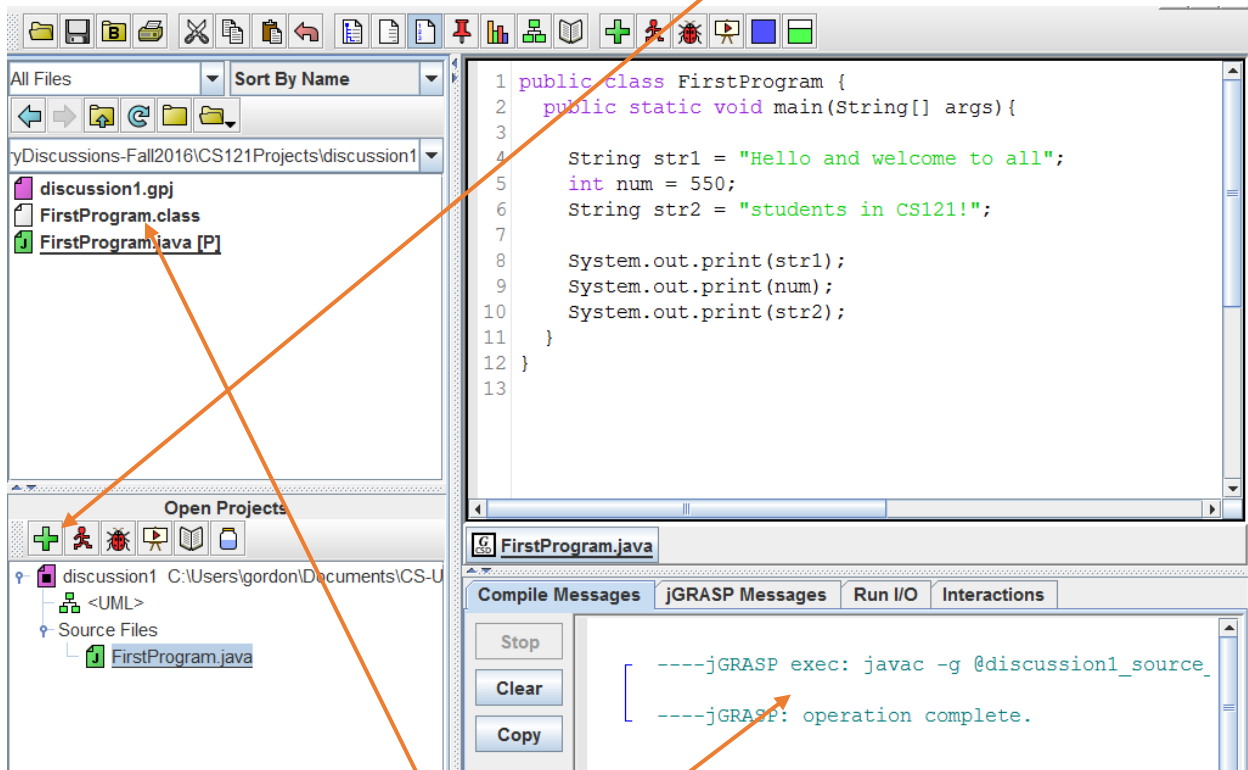


### Line numbers:

Line numbers are very useful when reading code. If you don’t see line numbers, click on the “View” menu, select “Line Numbers” option. Notice the appearance of line numbers in the editor pane. These numbers are very useful for debugging and troubleshooting.

To this point, you have created a project, added a Java source-code file to it, and have opened that file for editing in jGrasp.

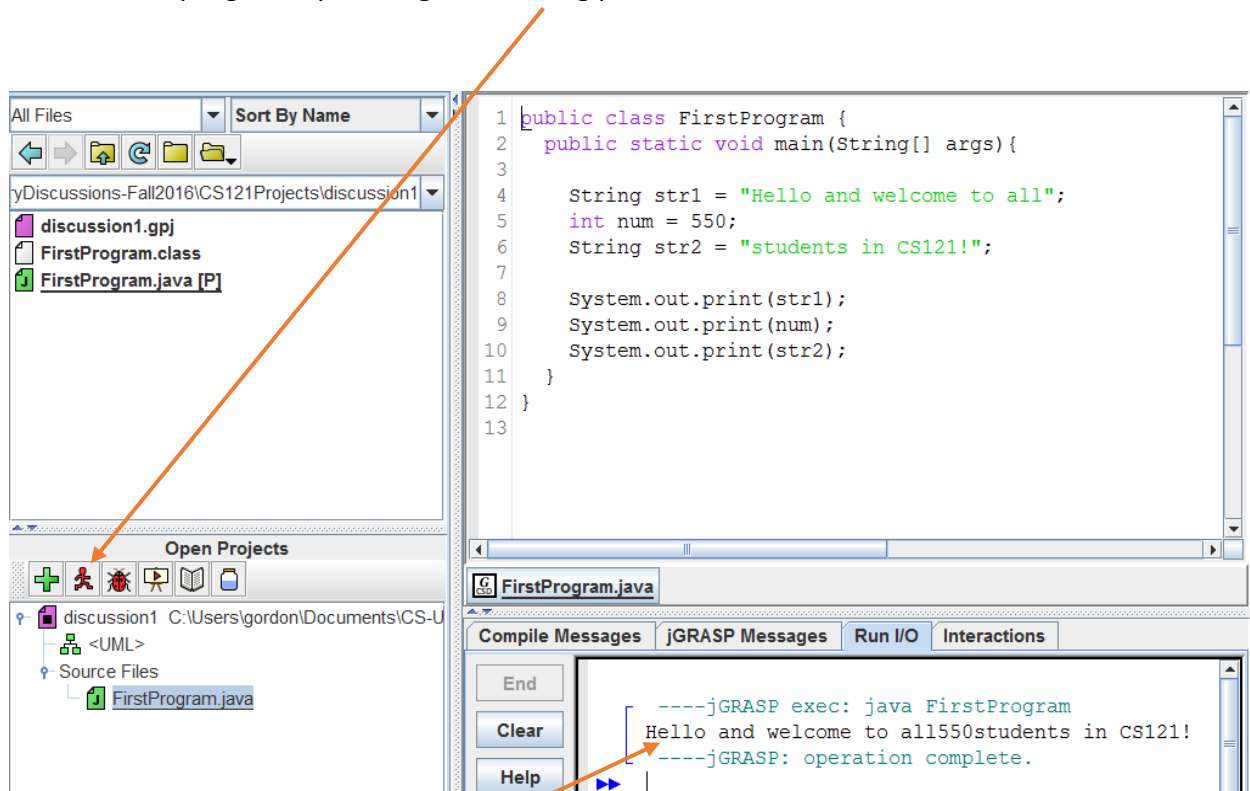
The next step is to compile the program. Click on the green cross in the project pane:



Note the messages of successful compilation in the interactions pane. Also note that the compilation produced the file "FirstProgram.class". This is the file that Java will run.

The compilation of the source code file "FirstProgram.java" is carried out by the java compiler, "javac". This produces an "object" file, "FirstProgram.class". Note that changes you make to the source code file are automatically saved when you compile.

Now run the program by clicking the running person icon.



Notice the message printed to the interactions pane (or, “console”). The jGrasp run button invoked the Java runtime environment to execute the main method in the FirstProgram class file. This code was ultimately executed by the Java Virtual machine (JVM) which communicates with the computer’s operating system. Note that all computer resources are managed by its operating system. Java applications run “on top” of the operating system.

This is a summary of the development sequence:

Design/Develop Time: Create or modify a source code file or files.

Compile Time: Compile the source code into .class files. These can be executed.

Run Time: Execute a “main” method in a class. The program is run by the computer.

### The main method:

A Java program must have at least one method called “main”. This is the point from which the code is executed at run time. A program may have many main methods. The program can be run from any main method. The term “method” is used in Java (and in Object-Oriented programming in general) to mean “function”.

The syntax of the main method is:

Return type      Method name      Parameter: Data type, name of parameter

```
public static void main(String[] args){
    // code goes here
}
```

Enclosing curly braces- the "scope" of the method.

You do not need to worry about what all these words mean now as we'll cover that later. The main points to consider now are that a function, as in mathematics, has a name, takes input in the form of zero to many parameters (enclosed as a comma-delimited list in parentheses), and results in some actions that may result in an output value. In Java, all code that is executed by a method is enclosed in curly braces { }. A set of opening and closing curly braces defines a block of code, called a "scope".

### The source code file:

Java code must be in a plain text file with a .java extension. In the above example, the file is called "FirstProgram.java". This is called a source code file. Java programs are made up of one or more source code files. A Java program can have hundreds (or more) of source code files in its code base.

Every source code file in a Java application is a Class definition. A Class is the basic "unit" of a Java program. A Class is meant to contain, or "encapsulate", a set of related data and functionality. For example, a Calculator class would contain numeric data and mathematical methods. All Java code must be located inside of a Class definition. The difference between an Object and a Class will be covered soon. For now think of a Class is a sort of "container" that holds all of the Java code for a program.

The syntax of a Java Class definition:

Class name

```
public class FirstProgram {
    // class code goes here
}
```

Enclosing curly braces - the "scope" of the class.

Notice the name of the file must agree *exactly* with the class name- case sensitive. For example, if the above file was named "firstProgram.java", the file would not compile. The Java compiler would output an error that the file name must match the Class name: "FirstProgram".

All of the code in a .java file must be located within the Class definition's curly braces.

### Commenting code:

Comments may be added to Java code in two ways: a double forward slash, //, will allow you to enter text on one line which the compiler will ignore and will not be executed at run time. JGrasp highlights comments in orange as you can see in the examples above. For multiple line comments, use the following symbols to begin a commented area: /\*, and the reverse \*/ to end. For example:

```
// single line example

/* multiple line
comment
example.
*/
```

Use comments sparingly. Comments should be used when there is a need to explain something important that isn't obvious to another developer. We'll provide more examples of good commenting later on.

### Indenting code and coding best practice:

Indent your code so it is easy to read and understand. Indenting means adding leading spaces to each line of code to make it easy to read and understand:

```
1  /*
2     This class prints a message to
3     the console via System.out
4  */
5  public class FirstProgram {
6      public static void main(String[] args) {
7
8          String str1 = "Hello and welcome to all";
9          int num = 550;
10         String str2 = "students in CS121!";
11
12         System.out.print(str1);
13         System.out.print(num);
14         System.out.print(str2);
15     }
16 }
```

Outer block, or scope

Inner block, or scope.

Indent according to the "block" created by opening and closing curly braces.

Don't cram a lot of statements into a little space. Sure, the compiler doesn't care, but other developers will.

Development best practice is to write code so that another person can easily read and understand it. Software development is done by many individuals working on the same project over a period of time, not by a single person. Once a program has been built and released, others will be maintaining it and updating it. In "real life" you can guarantee that you will have to work with code that someone else wrote, and that others will have to work with code you wrote. Sloppy, poorly written code is difficult to understand, is unprofessional, and results in lower productivity. Write code with the idea that another person will be reading it. (In this class, the instructor, TAs, tutors, and graders will be reading your code).

### Statements in the code:

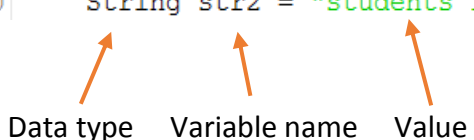
The code in the "FirstProgram.java" file consists of several statements defined inside the scope of the main method. The main method is defined in the scope of the FirstProgram class definition. Note that scopes can be nested inside of each other. Remember that a scope is defined by opening and closing curly braces , { }.

There are two types of statements in the code: assignment statements, on lines 8,9,10, and input/output (I/O) statements on lines 12,13,14.

Assignment statements "assign" a value to a variable. Variables must be "declared" with a data type and a name. Then, the equals sign, =, is used to assign a value to the variable. The values are stored in computer memory. Think of the variable as an address to the memory location that stores the corresponding value assigned to that variable.

Data types can be a "String" (has to have a capital S), a sequence of characters, or a number, either an integer, "int", or a decimal number, "double". The variable is always on the left-hand side of the equals sign and the value on the right hand side.

```
8   String str1 = "Hello and welcome to all";
9   int num = 550;
10  String str2 = "students in CS121!";
```



Data type    Variable name    Value



Choose meaningful names for your variables. Avoid using variable names such as “x” or “a”. Good naming helps communicate what the purpose of the variable is in your program. This helps others (and you) more easily understand your code.

The I/O statements (input/output statements) use the “System” Class to write the values assigned to the variables to the console- the interactions pane in jGrasp:

```
12 System.out.print(str1);  
13 System.out.print(num);  
14 System.out.print(str2);
```

The System Class receives the variable as a parameter (to its “print” method). Java “looks up” the value assigned to that variable. Then, the System Class asks the operating system to print that value to the console in jGrasp. Note that the integer, 550, gets changed to a String type by the System Class so it can be appended to the other Strings. We’ll discuss how this works in more detail later on in the course.

### **Make a change to the program:**

Notice that the message printed to the console is:

```
[ ----jGRASP exec: java FirstProgram  
Hello and welcome to all550students in CS121!  
----jGRASP: operation complete.
```

This is not great as there should be a space before and after the 550. A space is a character just like any other letter or symbol. Add a space to the end of the String value that is assigned to str1 and to the beginning of the String value assigned to str2. Recompile and print out the result. You should see:

```
[ ----jGRASP exec: java FirstProgram  
Hello and welcome to all 550 students in CS121!  
----jGRASP: operation complete.
```

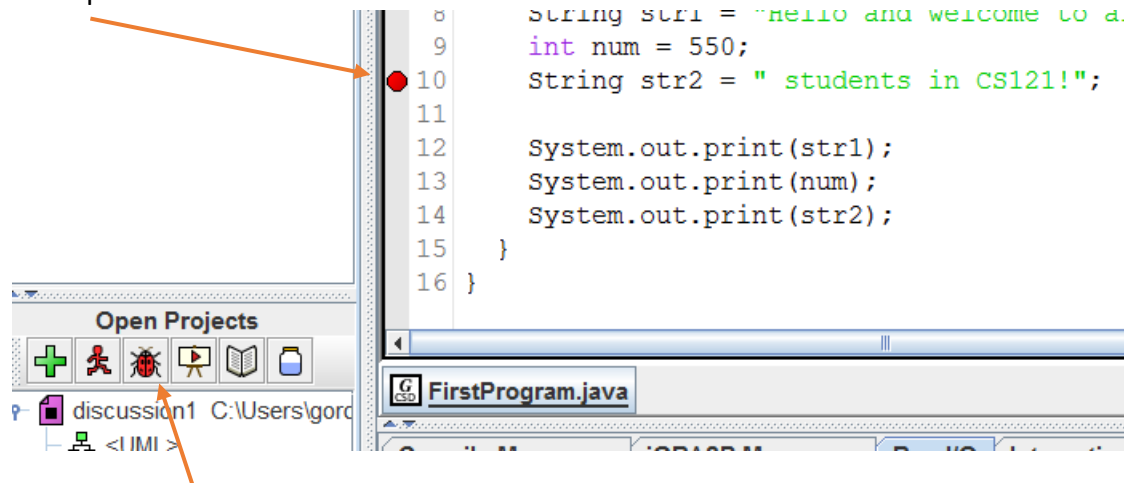
### **The Debugger:**

A debugger is an application that allows you the developer to have step by step control over the execution of a program, and to see the values of variables during the execution. Next, you will practice using the basics of the jGrasp debugger. This is essential for developing the Programming Projects for this course.

The debugger will allow you to control the program's execution. Normally, the code gets executed line by line. When you run in debug mode, the code will be executed line by line until it hits a "breakpoint". When a breakpoint is hit, the execution pauses until you command the debugger to continue. You can set any number of breakpoints in your code.

Now you'll set a breakpoint at line 10 and run the debugger. To set the breakpoint, move the cursor over the grey vertical line on the left hand side of the editor pane. You will see a red "stop sign", a red octagon, appear. Move the cursor until it is next to line 10 and click. The red octagon should stay put.

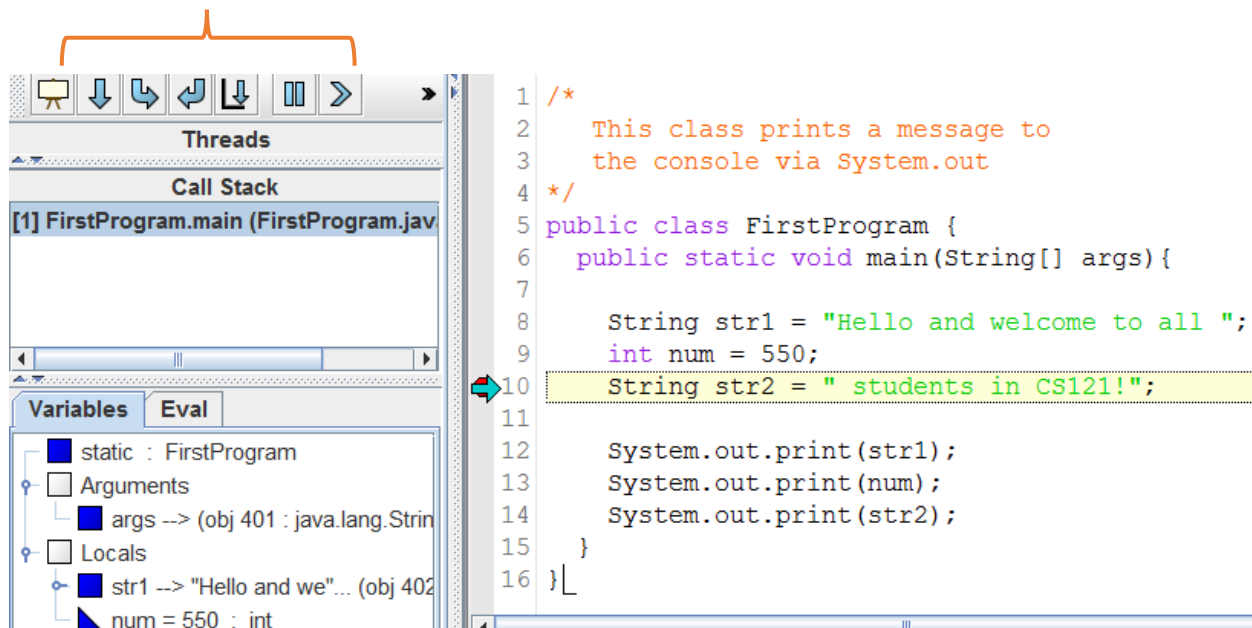
Breakpoint



Click the red bug icon to run the debugger.

The program will execute all of the code up to the breakpoint, and then pause on line 10. (The statement on line 10 will not have been executed). You will see a blue arrow appear over the red octagon, and the statement on line 10 will be highlighted. There are some other changes to the jGrasp interface. The "Variables" tab in the project pane is now active, and shows all of the variables that have been declared as well as their current values. This is a highly useful feature when developing and troubleshooting code.

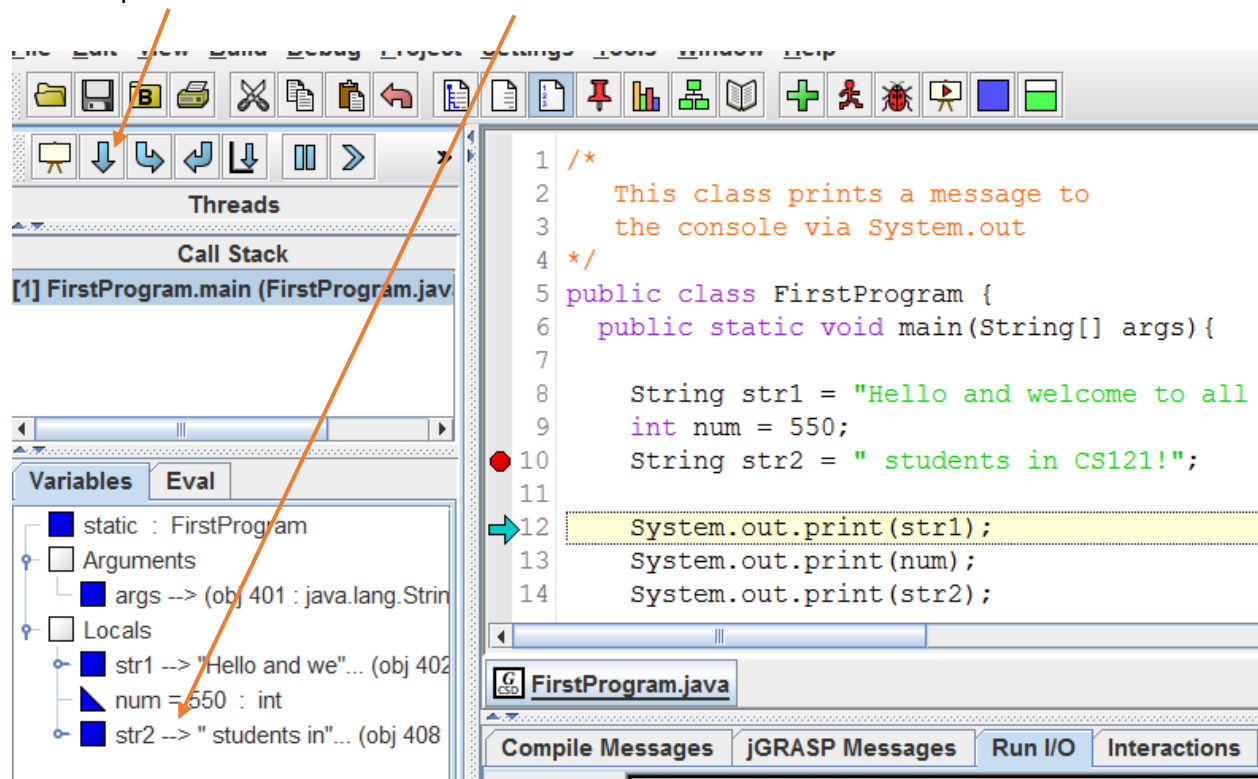
## Debugger controls



Notice that the variable “str2” does not appear in the Variables pane. That is because it is declared on line 10, which has not been executed yet.

Now, execute line 10 by clicking once on the blue, downward pointing arrow. This will “step” the debugger once- executing a single line of code. After clicking the down arrow, you should see the blue arrow move to the next line of code, line 12, and stop. Note that nothing will have been printed yet as line 12 has not been executed yet. You should see this in jGrasp:

The “step” arrow. The variable “str2” has now been created and initialized.



Now click the “step” arrow several times until the program ends. Watch how each statement prints to the interaction pane.

## Compile errors

What happens when there are errors in the code and it won't compile? Change the statement on line 12 so that it begins with a small s: system.out.print(str1). What happens when you try to compile? You should see this in the output pane:

```
----jGRASP exec: javac -g FirstProgram.java
FirstProgram.java:12: error: package system does not exist
    system.out.print(str1);
    ^
1 error
```

This is an error from the Java compiler. You get several pieces of information here. First, you see the file where the error occurred (we only have one file here- FirstProgram.java), followed

by the line number, 12 in this case (one good reason to enable line numbers!). Then, you see the error message- the compiler doesn't know what "system" is. On the next line you see the statement that the compiler didn't like. Note the little caret under "system". Some compiler messages are not so easy to decipher. For example, the word "package" may not mean anything to you. Don't worry, just get what information you can from the error message. You'll be getting plenty of opportunities to investigate errors and how to debug code later on. Now fix the problem and re-compile. Note that you can return to the "Project view" by clicking the "Browse" tab.

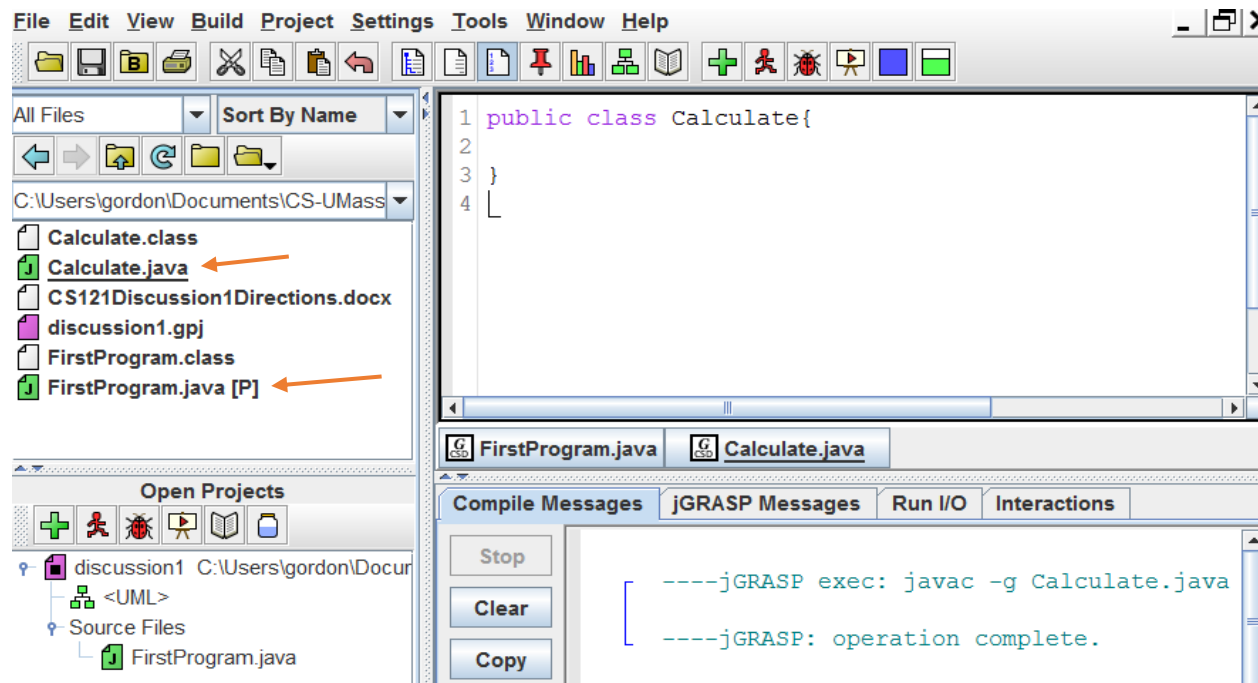
## **Part 2: Creating a new Java file.**

Now you'll create a new Java source code file from scratch and add it to your project. Go to the "File" menu, select "New", then "Java". A new edit window opens. You will create a simple Java program that calculates arithmetic operations and prints the results to the console. The class is called "Calculator". Type the following class definition into the new window:

```
public class Calculate{  
  
}
```

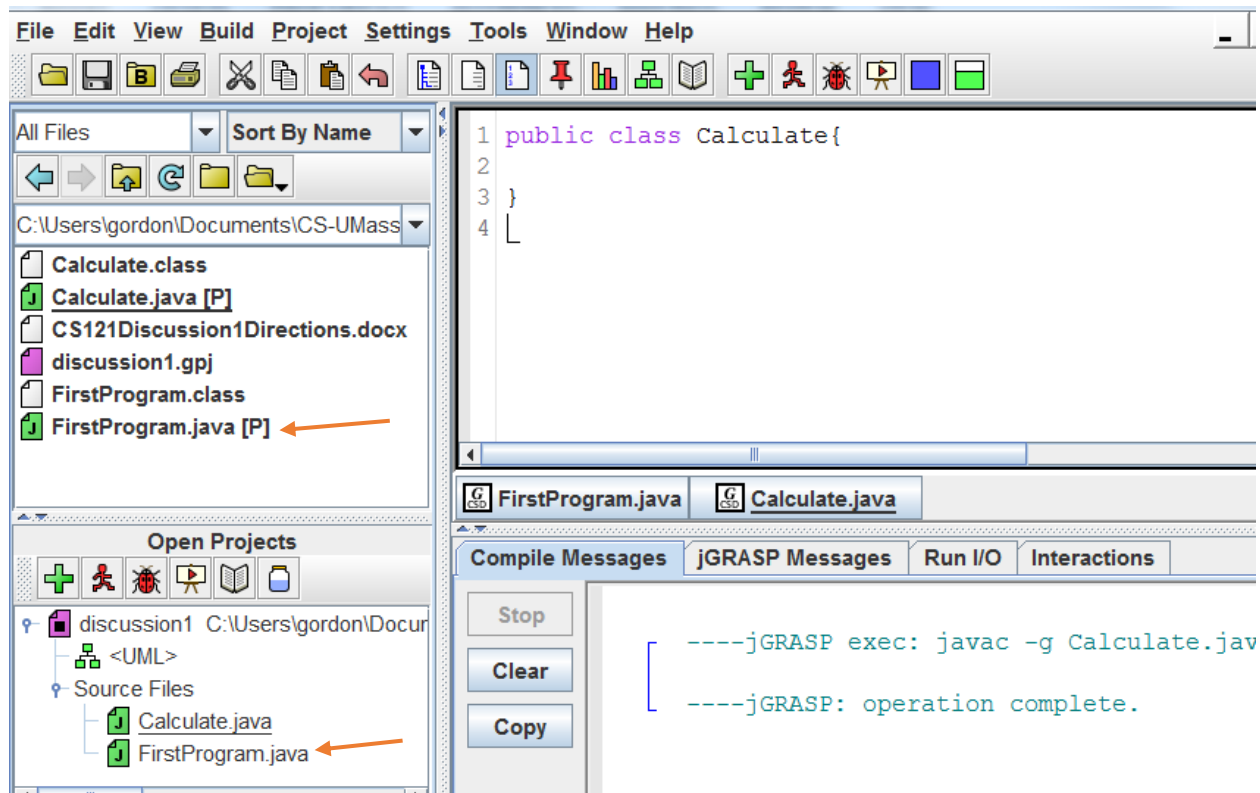
Click on the compile icon (the green cross). You will see the save dialog box. Click on the "Save" button. A file browser is displayed at the directory shown in the navigation pane (should be "discussion1"). Notice that the file name field at the bottom is already populated with "Calculate.java". Click "Save". You should see that the compile went through ok:

The new file has been created, saved, and compiled, but it is not part of your project yet:



You can see this because the Calculate.java file appears in the Navigation pane but not in the Project pane. You also see that there is no [P] after the Calculate.java file in the Navigation pane.

To add the new file to the project, click on the “Project” menu, “Add Files”. Select the “Calculate.java” file, click “Add” then “Done”. You’ll see the file has been added to your project:



## Writing Java statements.

We want to be able to run the Calculate class as its own program, so add a main method. Use the main method definition from the FirstProgram.java file. (copy and paste line 6). Don't forget to also add a closing curly brace to close the scope of the main method.

Now, enter the following statements into the body of main:

```
int valOne = 5;
int valTwo = 20;
int myResult = valOne + valTwo;
System.out.print("result: "+ myResult);
```

Compile the code. These assignment statements declare and initialize three integer variables named "valOne", "valTwo", and "myResult". Remember that assignment statements have a left hand side, LHS, and a right hand side, RHS, and use the "=" sign as an assignment operator. The assignment operator "=", does not mean equality, rather; it means that the value on the RHS is assigned to the variable on the LHS. The LHS has to be a variable, and the RHS has to resolve to a value. Note that the RHS can include variables.

For example, this statement is not legal:

```
25 = valOne;
```

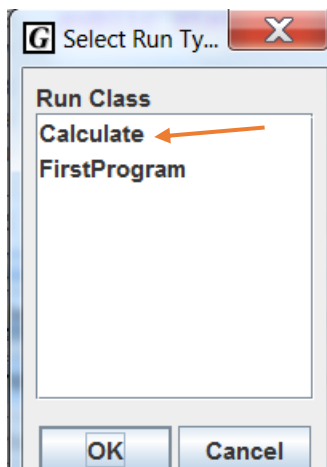
This one *is* legal:

```
valOne= 2*valTwo;
```

Back to the code in our Calculate class. The first two assignment statements assign the values of 5 and 20 to the variables `valOne` and `valTwo`, respectively. The variable `myResult` is initialized to the sum of `valOne` and `valTwo`. Finally, the value of `myResult` is printed to the console.

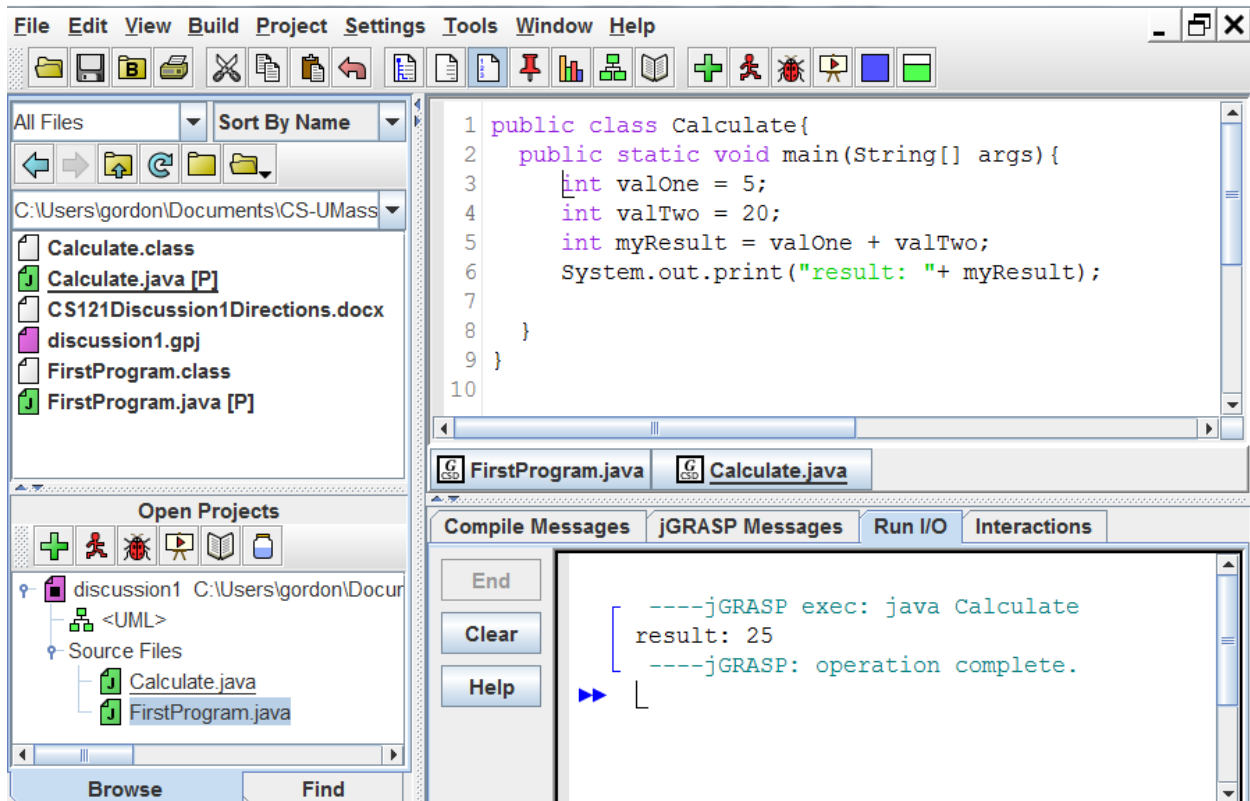
Note the use of the plus symbol, `+`, in the code. In line 5 it is used as the *addition* operator, and in line 6 it is used as the *concatenation* operator, where a String `"result: "` is concatenated with the variable `myResult`, which is an integer. More on data types and operators soon.

Now, run this code. Use the run icon above the project pane. Since there is more than one main method in your project, you see a dialog box asking you to select which main to run:

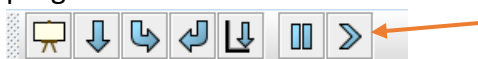


Select `Calculate` and click `"OK"`. You should see this in the output pane:





Now, set a breakpoint at line 5 and run the debugger. You see the variables `valOne` and `valTwo` under “Locals” in the Variables pane on the left. This pane shows you the values of all variables in your code- “local” means the local scope- the variables declared in the “local” curly braces. Now click on the blue, down arrow to step the debugger to line 6. Notice the appearance of the variable `myResult` and its value in the debug pane. Click the chevron, `>`, in the debug controls menu. This will run the program until the next breakpoint. In this case, there are no more so the program will run to the end.



Now insert these two lines between lines 5 and 6:

```

int valOne = 7;
int myResult = valOne + valTwo;

```

So you now have:

```

int valOne = 5;
int valTwo = 20;
int myResult = valOne + valTwo;
int valOne = 7;
int myResult = valOne + valTwo;
System.out.print("result: " + myResult);

```

Compile. Got errors? You betcha! Try to see what the problem is.  
The fixed up code should look like this:

```
int valOne = 5;
int valTwo = 20;
int myResult = valOne + valTwo;
valOne = 7;
myResult = valOne + valTwo;
System.out.print("result: "+myResult);
```

Do you see what the problem was? Variables may be declared only once within a scope. Compile and run in debug mode with a breakpoint set on line 5. Step twice so you are on line 7. Notice how the value of valOne changed- it's highlighted in red in the Variables pane. The valOne variable has been *re-assigned*. Step once more to line 8. Notice the change in the variable myResult. Run out the code (use the chevron, or click the step arrow until done).

#### Decimal numbers:

Now, we will explore the combination of integer and decimal numbers in Java. Integers are whole numbers, such as -5, 0, 33, etc. Decimal numbers have an integral part and a decimal part. For example, 23.009 has the integral part 23 and the decimal part .009. In Java, these numbers have the corresponding data types "int" and "double". The term "double" stands for "double precision", where precision is the number of decimal places to the left that can be represented by that data type. Computers are limited in the precision of decimal numbers they can handle by their hardware. Note that an integer is less "precise" than a decimal number. For example, 23 is not as precise as 23.009.

What happens when int and double numbers are combined?

Enter the following statements in the code above just before the System.out.print statement on line 8:

```
int myIntVal = 25;
double myDbIVal = 5.009;
```

Then, on the next line (line 10) write an assignment statement that assigns the sum of myIntVal and myDbIVal to the variable myResult. What happens when you compile this code? Can you explain the error message?

The variable myResult is data type int, which has no decimal places. One way to fix this is to change the data type of the variable myResult from an "int" to a "double".  
Do this (on line 5), recompile and run.

## Division in Java:

In Java you have to be careful with data types when performing division and other arithmetic operations.

In the real world, dividing 25 by 7 results in a decimal value, approximately 3.57. To implement this division in your program, change line 10 so that myResult is assigned to the division of myIntVal by valOne:

```
myResult = myIntVal/valOne;
```

Compile and run. The value of myResult is printed as 3.0:

```
[ ----jGRASP exec: java Calculate
  result: 3.0
  ----jGRASP: operation complete.
```

What has happened? Remember that Java always resolves the RHS of an assignment statement into a single value before it does the assignment. Since both myIntVal and valOne are integers, Java did the division and chopped off the decimal part to get 3. Then, when it assigned this result to myResult, it “cast” the 3 into a double since myResult is a double data type. Casting means changing one data type into another. When 3 is cast to a double it becomes 3.0. That is the result you see printed. The system as well as a programmer can make casts.

To get the correct decimal result, you need to cast either the numerator or denominator to a double. This is done by using parentheses:

```
myResult = (double)myIntVal/valOne;
```

In the above statement, the numerator is cast to a double, and the division will be a double value; Java will not “chop off” the decimal part. Make this change to line 10, compile and run. You should see:

```
[ ----jGRASP exec: java Calculate
  result: 3.5714285714285716
  ----jGRASP: operation complete.
```

Notice the precision is to 16 places!

You now have created a project with two Java programs. You are well on your way!