

COP2800C Module 5 Practice Exercise

In this practice exercise we will continue revising our box factory application in order to practice the following new concepts from this module:

- increment operators
- wrapper classes
- checking equality
- ternary operators

In addition to using these new features we will practice creating and executing JAR files. JAR stands for **Java Archive**, a file format used for aggregating many files into one. It is a common distribution format for Java applications and libraries that you need to be familiar with. We will bundle our source and class files into a JAR file that can be executed from the command line. You can read more about JAR files here:

<https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html>

We will be working with the jGrasp IDE to code and test our source changes, and to create our JAR file. We will then be working from the command line to execute it. As with previous exercises, I **highly recommend** that you use the Horizon system to do this since everything is preconfigured, including the required execution search paths.

- Start by making the following changes to the Box class:

Add an equals method. As described in the lecture slides, the == operator can be used to compare primitive data types (e.g. int, double, boolean) but should not be used to compare two objects (including Strings). Using == will compare the memory addresses of the two objects, which is not necessarily what we are looking for when we do a comparison. Instead, we can override the equals method inherited from the Java Object class. This is similar to what we did with the toString method. By providing a customized equals method we as the developers can decide what constitutes equality between objects. In the case of our Box class, we will consider two boxes equal if they have the same color.

```
// check if boxes are equal
@Override
public boolean equals(Object o) {
```

Notice that the parameter of this method is **Object**, not Box. This is required since the parameter types of an overridden method must match the base class's method exactly, and since we are overriding the Object class's equals method, this is the same type declared by that class (take a look at the Object class's API doc page, find the equals method on that page to verify this for yourself: <https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>

Using Box instead of Object as the parameter type for the equals method is a common error that many beginning Java programmers make.

Let's look at the code which implements the equals method in the Box class:

```
boolean result = false;

result = (o instanceof Box) ?
    ((Box)o).getBoxColor() == this.getBoxColor() : false;

return result;
```

For our equals method to be implemented properly, we first need to verify that the object passed in is indeed a Box. While it's not likely some other type of object would get to this point, we do this verification using the **instanceof** operator.

We are using the instanceof operator as the first part of a **ternary operator** as discussed in the lecture slides. This is an efficient implementation of an "if" statement that we will cover in the next module. Using this operator is equivalent to the following code:

```
if (o instanceof Box)    // in the ternary, this is (o instanceof Box) ?
    result = ((Box)o).getBoxColor() == this.getBoxColor();
else                    // in the ternary, this is the colon :
    result = false;
```

Once we have verified that we have a Box parameter, the decision of whether the objects are equal is being made in this statement by comparing the colors (using the == operator since the colors are enum types which evaluates to primitive integers):

```
((Box)o).getBoxColor() == this.getBoxColor()
```

Here we are using a **cast** operation (described in the lecture slides: this is **narrowing** cast) to cast the object o into a Box object; we are safe doing this because we have verified the passed object is a Box. The parentheses are a required to call the Box object's getBoxColor accessor (trust me when I say even the most senior Java developers sometimes have to play around with ordering of parentheses to get them correct). This comparison tests if the passed Box object color is equivalent to the calling object's color (also a Box) and sets the result variable to true, otherwise it remains false.

- Now let's move to the BoxFactory class. I have added a local Integer variable to the toString method and to the main method which acts as an array index. This is an example of the **wrapper** class discussed in the lecture slides. Integer is a class wrapper for the primitive int type:

```
Integer boxNum = 0;
```

Now each time I reference an element in the arrays I am incrementing this variable instead of using the numeric literals 0, 1, and 2 as index values. Note that I must make sure I reset the variable to zero whenever I start the increment operations over from the beginning again. For the increment operation I am using the postfix ++ operator which evaluates the value of the variable **before** incrementing it. The increment operation is actually performing the unboxing process described in the lecture slides. Here's the code in the toString method:

```
Integer boxNum = 0;
...
// append the boxes
retString += "First Box is a " + this.bboxes[boxNum++] + " box.\n";
retString += "Second Box is a " + this.bboxes[boxNum++] + " box.\n";
retString += "And the final box is a " +
            this.bboxes[boxNum] + " box.";
```

Note that when I access the third array element, I do not increment boxNum again since there are no more array references which follow. If I want to start from the first element, however, I must reinitialize the variable to 0.

I use the same strategy in the main method:

```
// local counter to keep track of number of boxes
Integer boxNum = 0;

// add boxes to the array
boxArray[boxNum++] = new Box(Color.RED);
boxArray[boxNum++] = new Box(Color.GREEN);
boxArray[boxNum] = new Box(Color.BLUE);
```

Here is where I reinitialize the variable:

```
// make some changes to test the mutator
System.out.println("Testing mutators and accessors");
boxNum = 0; // reset counter
boxFact.bboxes[boxNum++].setBoxColor(Color.BLUE);
boxFact.bboxes[boxNum++].setBoxColor(Color.RED);
boxFact.bboxes[boxNum].setBoxColor(Color.GREEN);
```

Be sure that you use these postfix increment operators in your submitted solution for this module. We'll use for loops later in the course (a much more efficient way to do this), but I want to see that you've mastered this concept first.

Now let's test the equals method. I only use the first element of the array for this, and compare it to a temporary Box object instantiated to do the comparison. First I use the == operator to show that the memory addresses of the objects are not the same – they are not the same object! But the the equals method is used to implement our rule that if two boxes have the same color, they are the same.

```
// create a new Box object and test for equality with first box in factory
Box compareBox = new Box();
compareBox.setBoxColor(Color.BLUE);

// verify first box in factory is blue
System.out.println("first box color: " + boxFact.bboxes[0].getBoxColor());

// compare with == (comparing memory addresses, should be false)
System.out.println("compare boxes with ==, result is " +
    (compareBox == boxFact.bboxes[0]));

// compare with equals method (comparing colors, should be true)
System.out.println("compare boxes with equals method, result is " +
    compareBox.equals(boxFact.bboxes[0]));
```

Here is the new expected output:

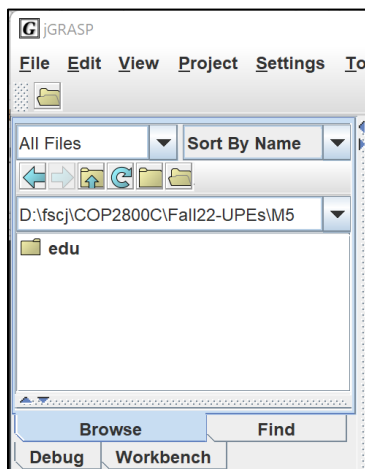
```
Box Factory contents:
There are 3 boxes in the factory.
First Box is a RED box.
Second Box is a GREEN box.
And the final box is a BLUE box.
Testing mutators and accessors
first box: BLUE
second box: RED
third box: GREEN
first box color: BLUE
compare boxes with ==, result is false
compare boxes with equals method, result is true
```

Once these changes have been implemented and the code has been thoroughly tested, we can look at creating our JAR file.

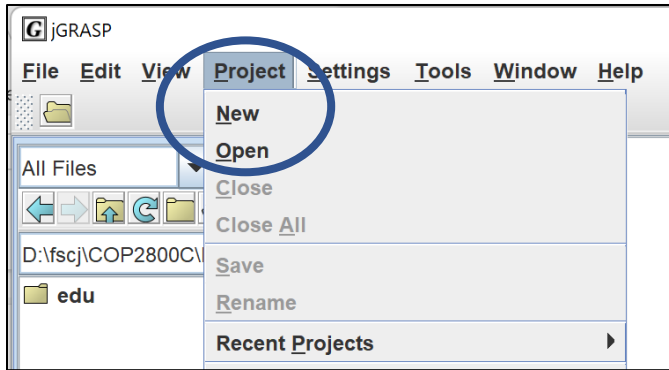
Creating the JAR File

Start by navigating to your top-level folder for your source code; my folder is named M5 so I am in the M5 folder, containing the source code folder structure edu\fscj\cop2800C\ with my source files down in the COP2800C subfolder.

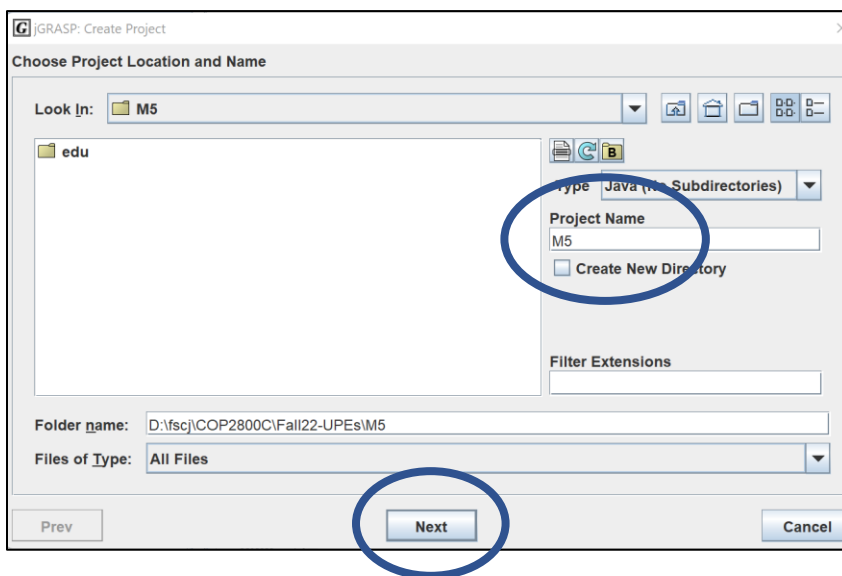
Open jGrasp and browse the that top-level folder as shown here:



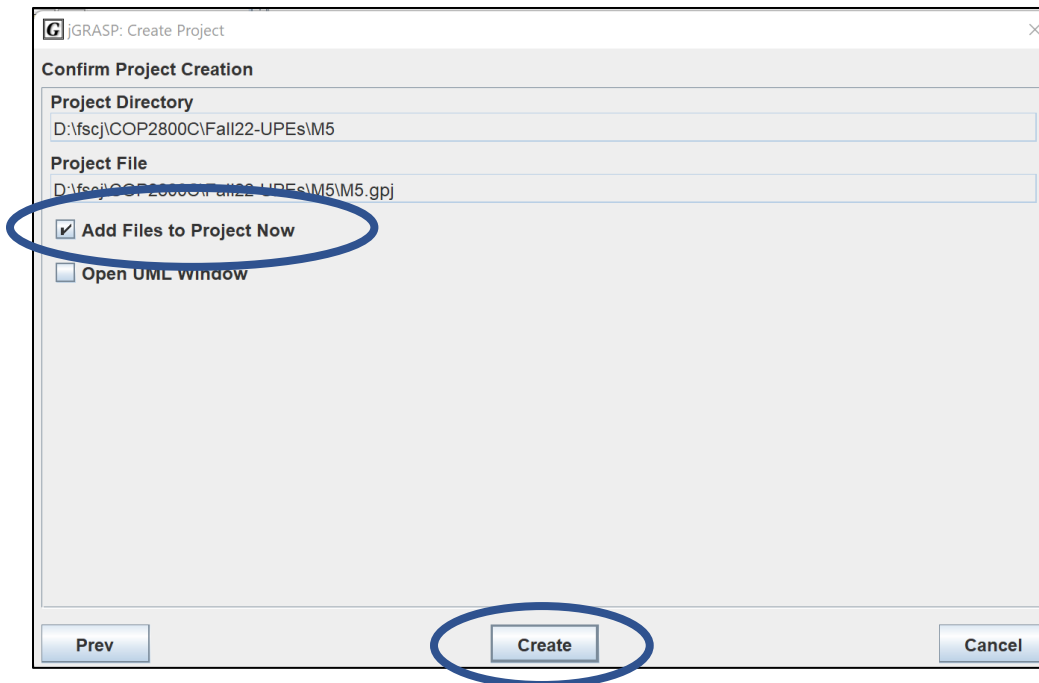
Now start a new project:



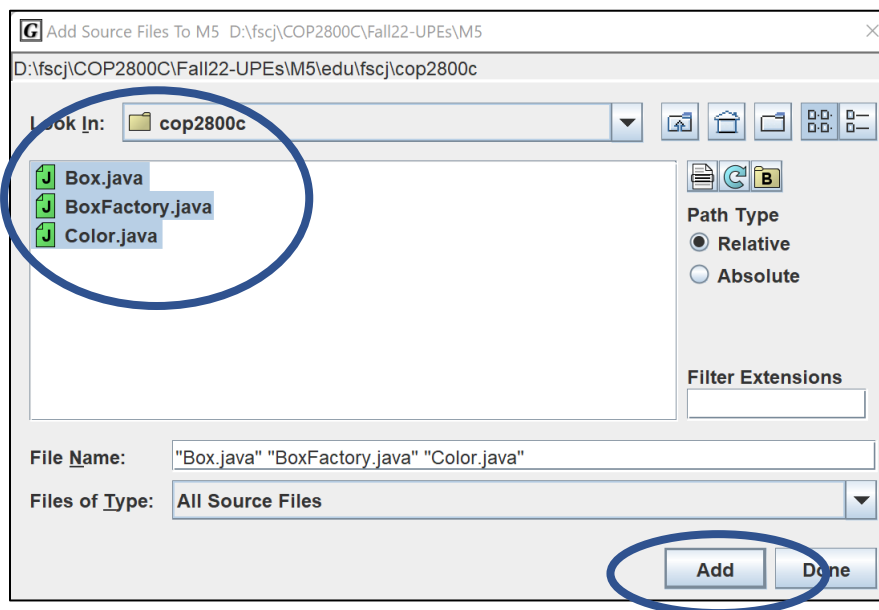
Choose and enter a project name, I have called mine "M5". Don't check any other options, and click Next.



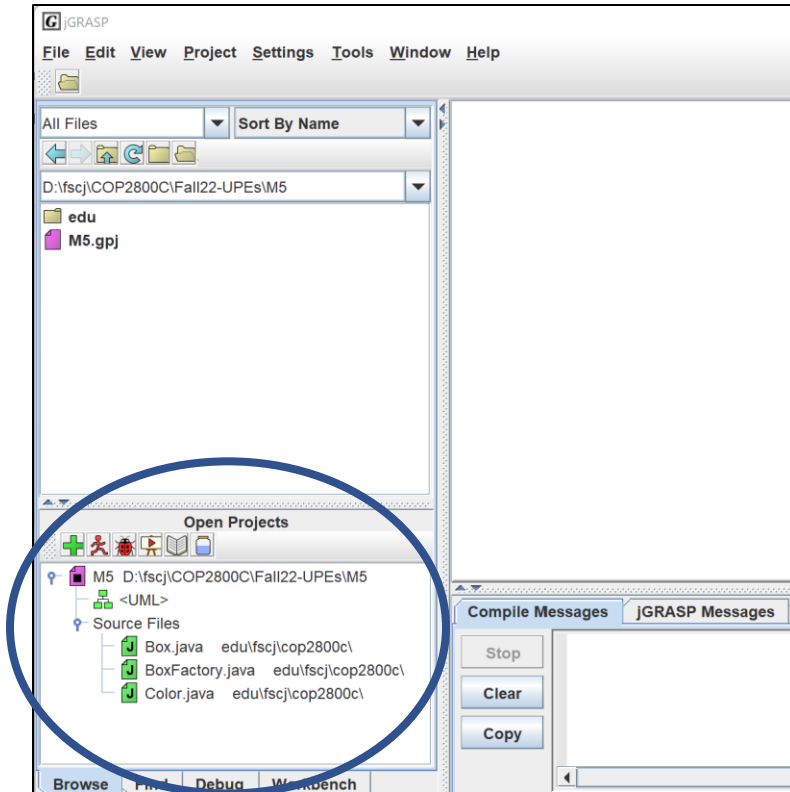
Verify your settings, check "Add Files to Project Now", and click Create:



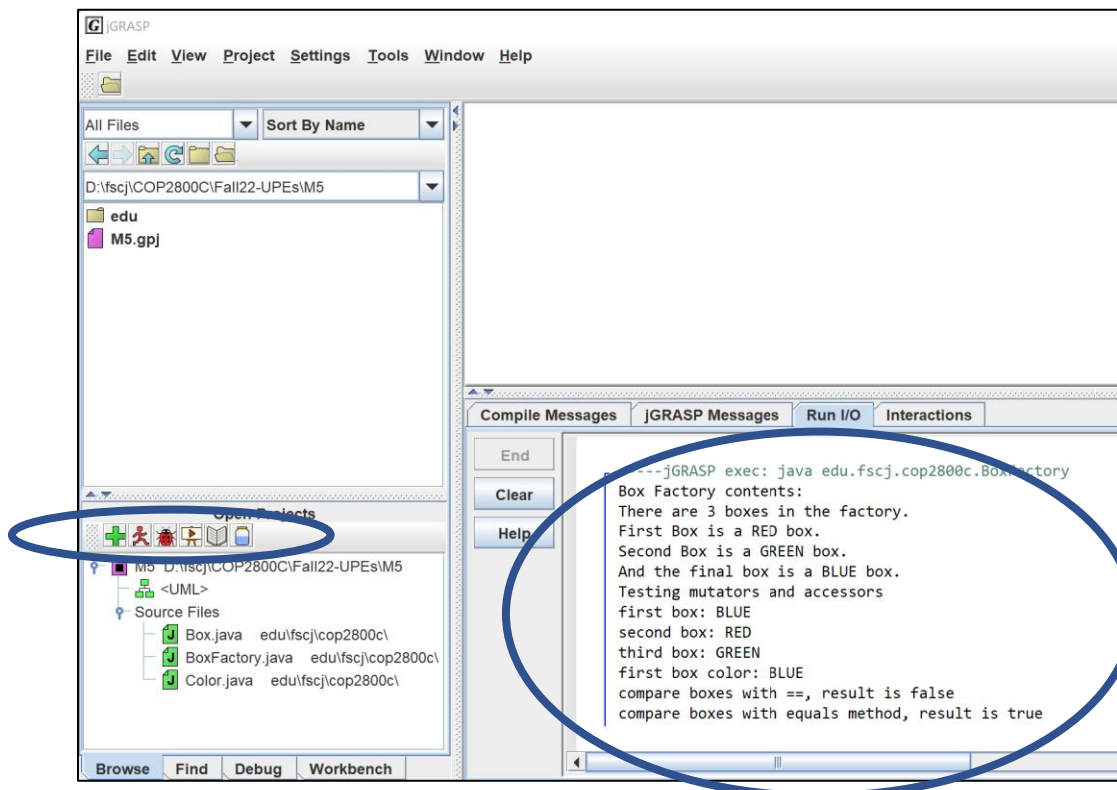
Double-click on the edu folder to navigate down to the bottom level (the cop2800c subfolder). Select all three Java files (select the top file, then shift-click the bottom file), click Add, then click "Done"



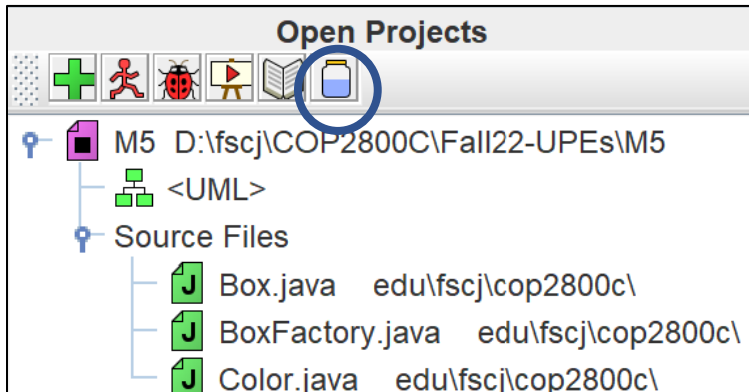
You should now see your project in the lower left pane of the jGrasp windows. Note that you have the same build tools in that pane that you do at the top of the IDE. These tools will build and execute your project.



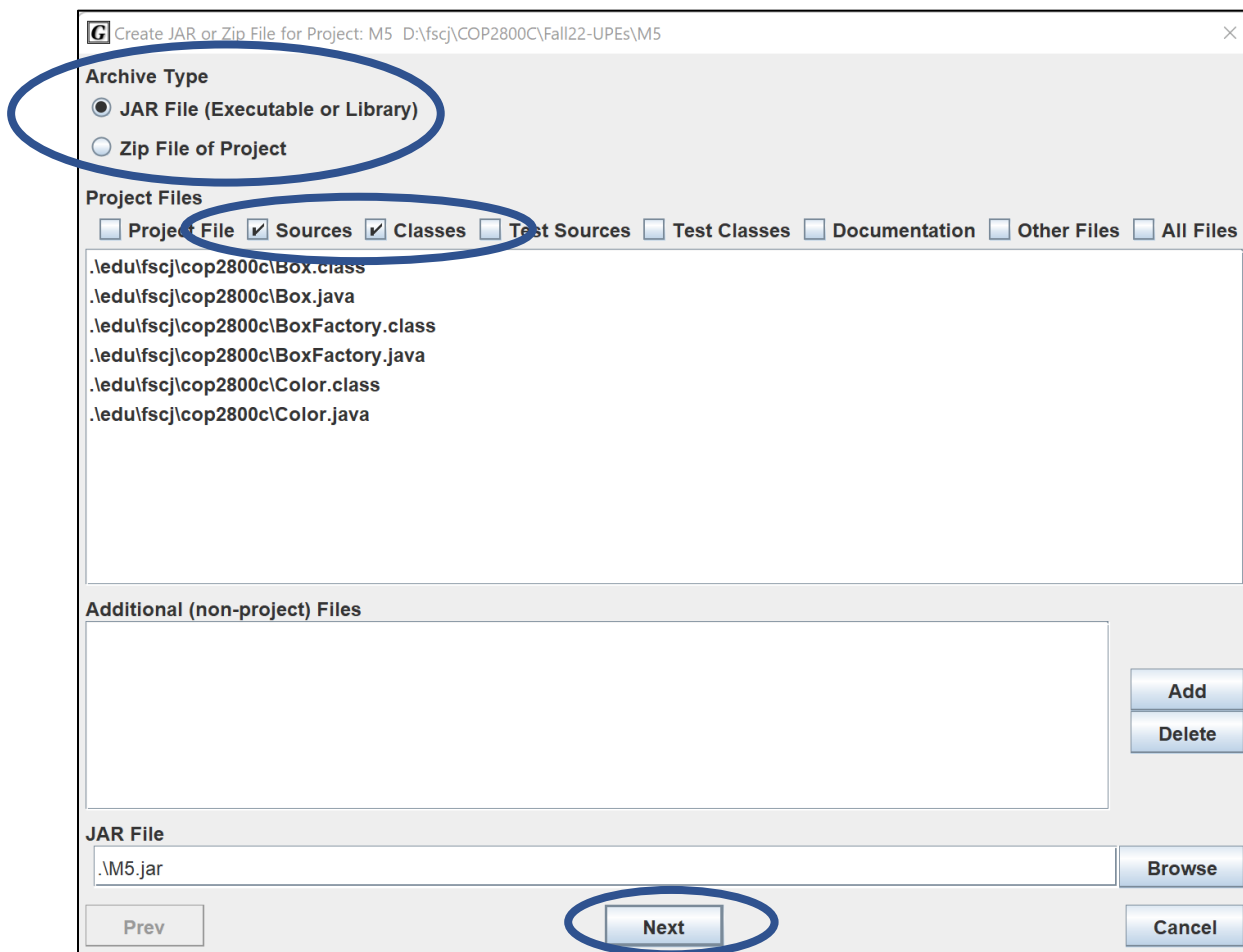
Go ahead and use the tools to build and execute to verify the project is configured correctly:



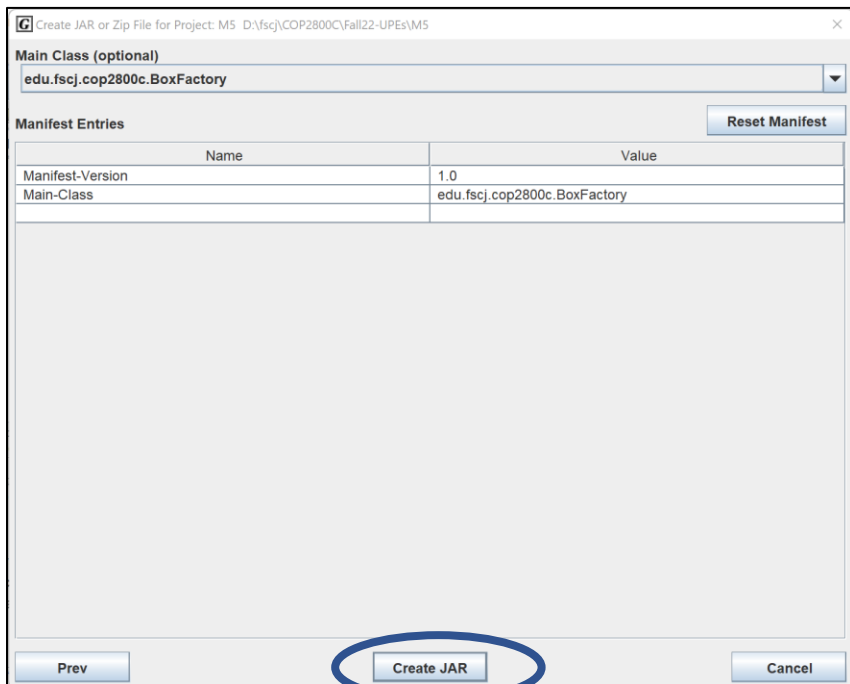
Now we will create the JAR file. Click on the jar icon on the right side of the project tool pane:



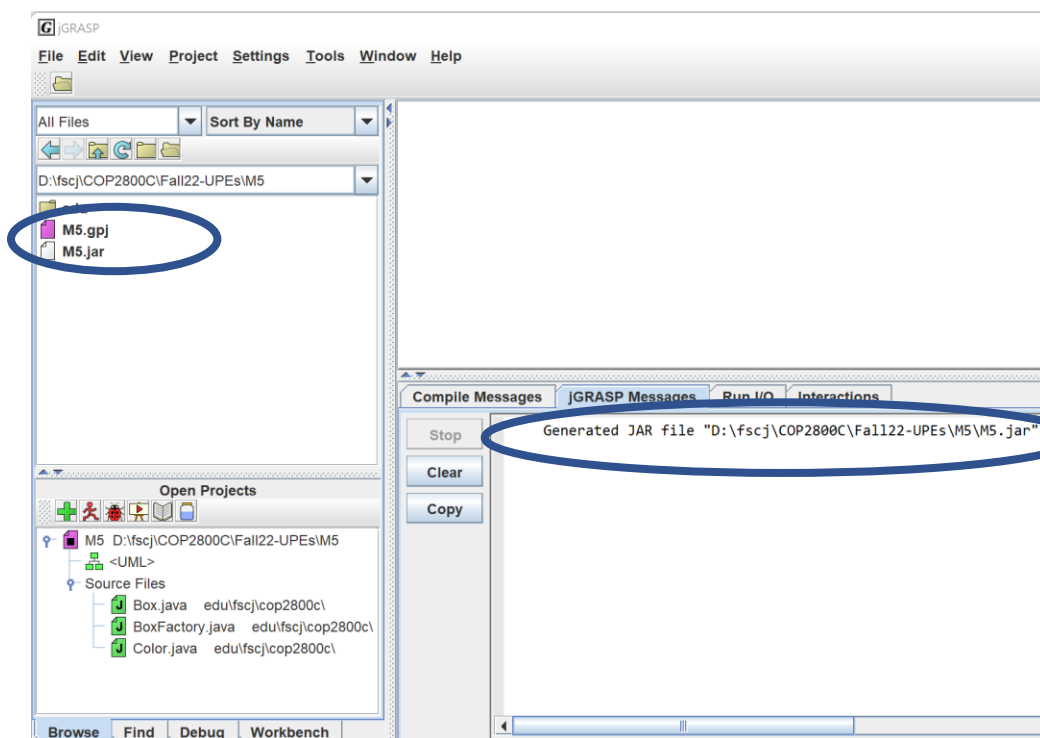
Click on the "JAR File" radio button, then check the "Sources" and "Classes" checkboxes. Do not check any other boxes. You should see your source files and your class files. After confirming this, click the "Next" button:



Here's the dialog which shows the "meta-data" for your JAR file. Notice that this dialog indicates that the main class for the application resides in the BoxFactory class, which is correct. There are many things we could add to this configuration, but we will just use the defaults for this assignment. After viewing the information in this dialog, click on "Create JAR".

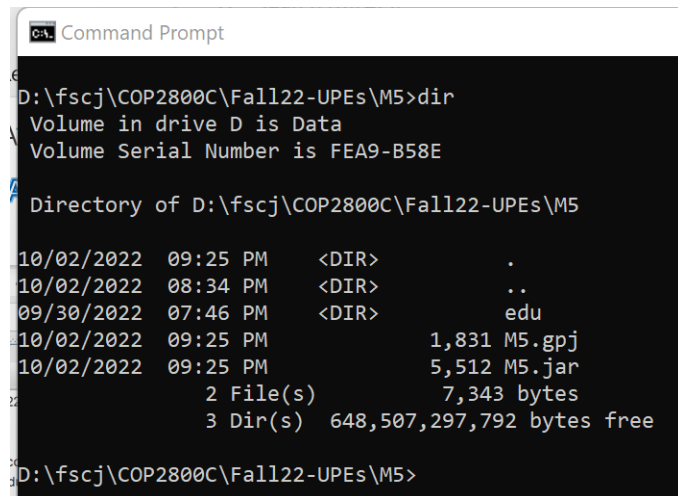


The File Browse pan should now show the JAR file in the list of files (the .gpj file is the jGrasp project file).



Now click File/Save All in jGrasp and exit the IDE if desired. Note that anytime you change a source file you will need to recreate the JAR file.

At this point you will need to bring up a Windows command tool application and navigate to the folder where your JAR file is located.



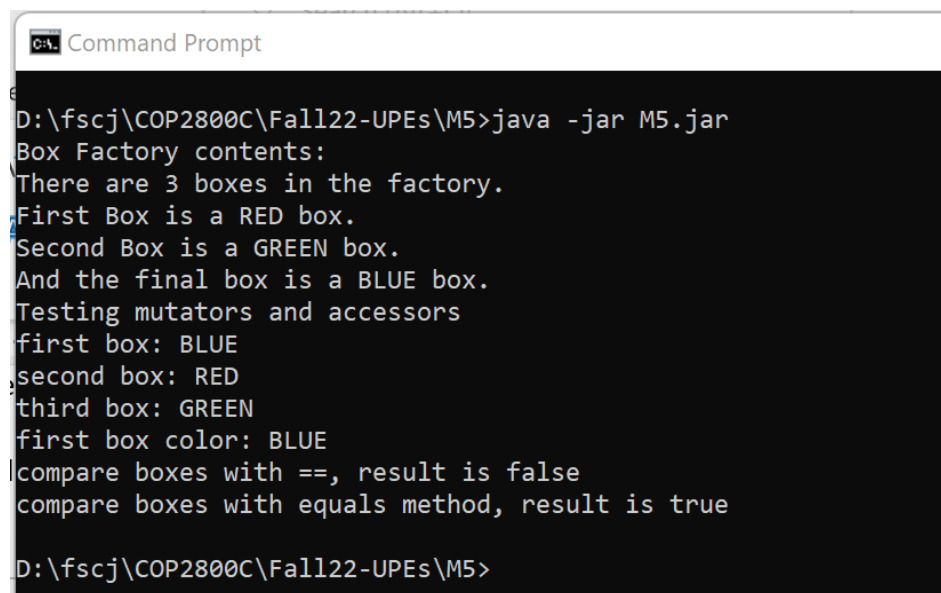
```
Command Prompt
D:\fscj\COP2800C\Fall122-UPes\M5>dir
Volume in drive D is Data
Volume Serial Number is FEA9-B58E

Directory of D:\fscj\COP2800C\Fall122-UPes\M5

10/02/2022  09:25 PM    <DIR>          .
10/02/2022  08:34 PM    <DIR>          ..
09/30/2022  07:46 PM    <DIR>          edu
10/02/2022  09:25 PM                1,831 M5.gpj
10/02/2022  09:25 PM                5,512 M5.jar
               2 File(s)              7,343 bytes
               3 Dir(s)  648,507,297,792 bytes free

D:\fscj\COP2800C\Fall122-UPes\M5>
```

All that is left now is to run the application using the jar file. We use the -jar option with the java command to do this:



```
Command Prompt
D:\fscj\COP2800C\Fall122-UPes\M5>java -jar M5.jar
Box Factory contents:
There are 3 boxes in the factory.
First Box is a RED box.
Second Box is a GREEN box.
And the final box is a BLUE box.
Testing mutators and accessors
first box: BLUE
second box: RED
third box: GREEN
first box color: BLUE
compare boxes with ==, result is false
compare boxes with equals method, result is true

D:\fscj\COP2800C\Fall122-UPes\M5>
```

We can now copy this JAR file to any system with Java installed and execute our application!