

Goals:

By the end of this project you should:

- Have an understanding of polymorphism via inheritance
- Be able to create methods that use polymorphism to deal with multiple types of objects

Directions:

Don't forget to add your Javadoc comments for your classes, constructor, and methods in this activity.

For this assignment, you will need your classes from Activity on "Inheritance" except for the driver program.

ItemsList

- Create a class called ItemsList. This class will hold an array of InventoryItem objects (which includes objects of subclasses of InventoryItem; recall an instance of a subclass of InventoryItem *is an* InventoryItem).
- Declare an instance variable in your class of type InventoryItem array which is called inventory and an int called count to track the number items in inventory.

```
private _____[] inventory;  
private _____ count;
```

- Create a constructor for ItemsList that has no parameters. Add the following code to instantiate the InventoryItem array.

```
_____ = new InventoryItem[20];  
count = _____;
```

- Create the following method stubs for your class:
 - addItem: with parameter InventoryItem itemIn; no return.
 - calculateTotal: with parameter double electronicsSurcharge; double return.
- The addItem method takes an InventoryItem as a parameter, assigns it to the element at position count in the inventory array, and then increments count. No need to worry about exceeding the capacity of the array in this activity.
- Create a toString method in ItemsList. The toString method should iterate through inventory from 0 up to count (but not including count) to append the toString for each item to output:

```
public String toString() {  
    String output = "All inventory:\n\n";  
    for (int i = 0; i < _____; i++) {  
        output += _____ + "\n";  
    }  
    return _____;  
}
```

Recall, the `Object` class has a `toString` method, which is inherited by its subclasses. When you define a `toString` method in your class, you are overriding an inherited `toString` method.

InventoryListApp – driver class and main method

- Create a class called `InventoryListApp` and add a main method. In the main method, instantiate a `ItemsList` object called `myItems`.
- In the main method, set the tax rate to 0.05 by invoking the `InventoryItem.setTaxRate` method.
- Instantiate the following 4 items in the main method and add them to `myItems`:
 - `ElectronicsItem`: name = “laptop”, price = \$1234.56, weight = 10 lbs
`myItems._____ (new ElectronicsItem(_____, 1234.56, 10));`
 - `InventoryItem`: price = \$9.8, name = “motor oil”
 - `OnlineBook`: price = \$12.3, name = “All Things Java”
 - `OnlineArticle`: price = \$3.4, name = “Useful Acronyms”
- Print the `toString` return value of the `ItemsList` object `myItems` to standard output:

```
----jGRASP exec: java -ea InventoryListApp

All inventory:

laptop: $1311.288
motor oil: $10.2900000000000001
All Things Java - Author Not Listed: $12.3
Useful Acronyms: $3.4

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

ItemsList: calculateTotal

- The `calculateTotal` method in `ItemsList` accepts surcharge for electronics items as a parameter and returns the total price of all of the items. You'll have to iterate through each of the items in inventory and add the cost for each item to a running sum (price below). If the item is an `ElectronicItem`, invoke `calculateCost` method and add the electronics surcharge that was passed as a parameter to `calculateTotal`.

```
double total = 0;
for (int i = 0; i < count; i++) {
}
return total;
```

- Use the *instanceof* operator to determine whether an object is an instance of **particular class** (place this *if-else* inside the for loop). If the object is of type `ElectronicsItem`, the following code shows how the *instanceof* operator is used to add the small surcharge to an `ElectronicsItem`:

```
if (inventory[i] instanceof ElectronicsItem) {  
    total += _____.calculateCost() + electronicsSurcharge;  
}
```

Add an else clause to calculate the cost without adding the surcharge:

```
else {  
    total += inventory[i]._____;  
}
```

- Be sure to return total after the loop.

Main Method

- In the main method, add code to print the total of all items and include an electronics surcharge of 2.0:

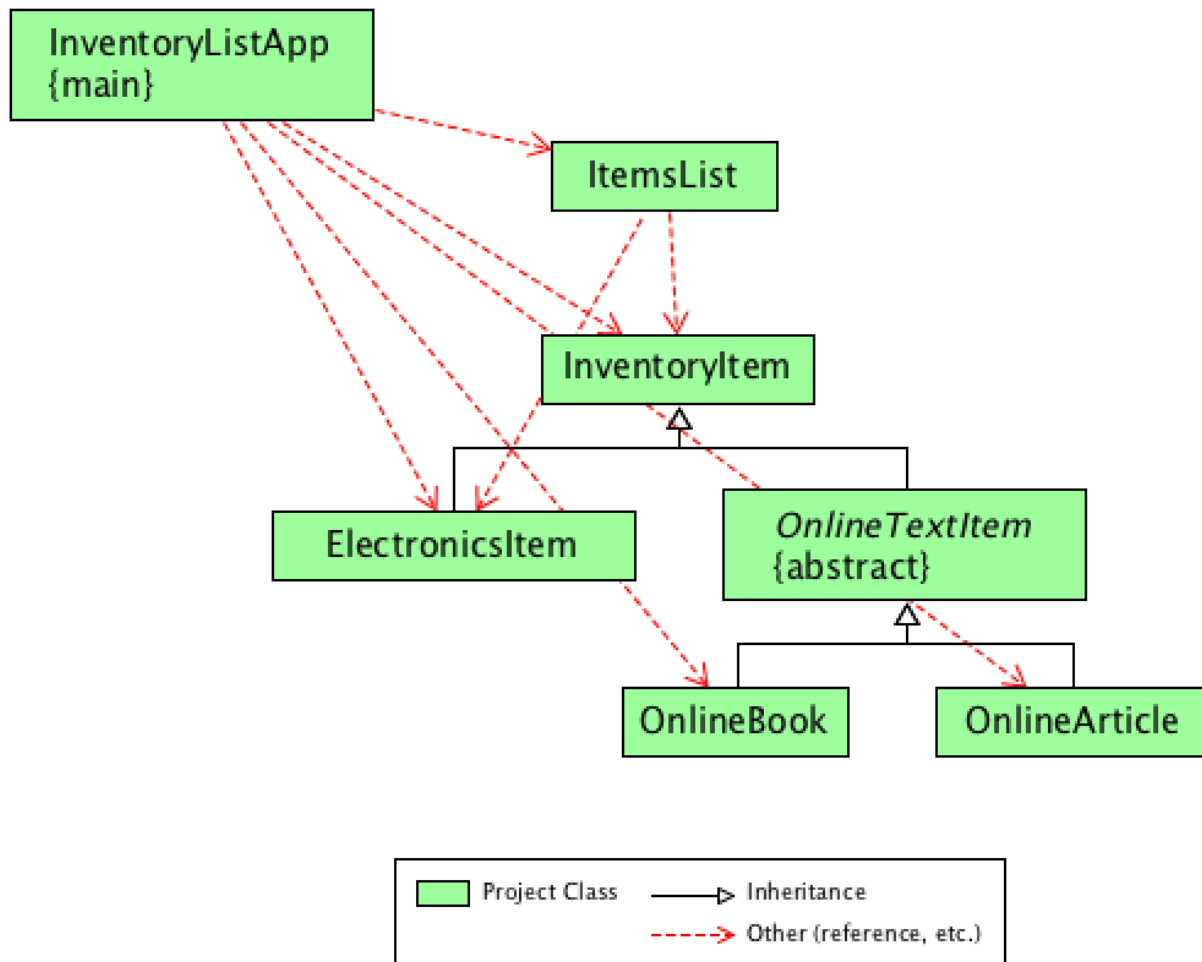
```
System.out.println("Total: " + myItems.calculateTotal(2.0));
```

- Your output should appear as follows.











```
----jGRASP exec: java -ea InventoryListApp  
  
All inventory:  
  
laptop: $1311.288  
motor oil: $10.290000000000001  
All Things Java - Author Not Listed: $12.3  
Useful Acronyms: $3.4  
  
Total: 1339.278  
  
----jGRASP: operation complete.
```

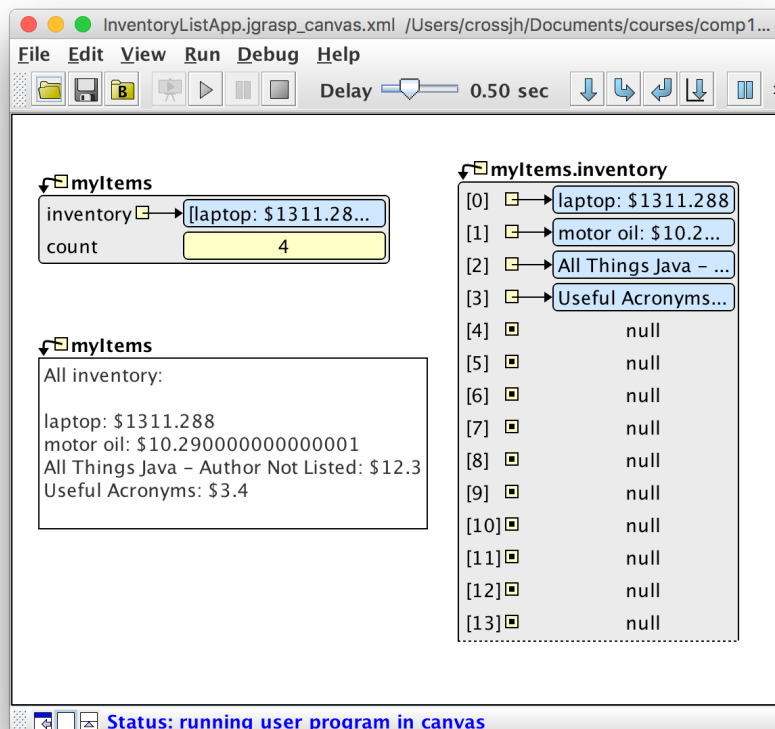
UML Class Diagram

- If you have not already done so, create a jGRASP project called InventoryListApp and add all of your classes for this activity.
- Generate the UML class diagram and arrange the classes as shown below.



Viewer Canvas

- Create a canvas for your program as follows:
 - (1) Run your program in the canvas . After the canvas opens, single step  your program until you see myItems in the Debug tab.
 - (2) Drag myItems onto the canvas. Then select the inventory field in myItems and drag it out onto the canvas.
 - (3) Drag myItems onto the canvas again and change the viewer to the “toString” viewer. To do this, select the viewer window, click the menu button  on the top-right of the viewer frame, then select “Viewer” then “toString”.
 - (4) Save the canvas then end the program. Run your program in the canvas . Now single step  your program and watch the objects appear on the canvas.
 - (5) Run your program in the canvas . Click the play button  and watch the canvas and stepping in the program as the objects appear on the canvas. Notice that the debugger is stepping into each of your constructors and method as they are called. You should be sure that you understand “how” your program is running to accomplish its task. Note that you can pause  the canvas and then hit play  to continue auto stepping-in. You can also set one or more breakpoints and then resume  to breakpoint.
- You should become comfortable using the canvas and debugger so that when needed, you will be able to use them to help find and correct errors in your projects.



- Finally, submit your files to Web-CAT.