

Preconditions and Postconditions

Purpose:

The purpose of this lab is to practice using method preconditions and postconditions, and to practice writing conditionals.

The class *Register* revisited:

In this lab, you will modify your specification and implementation of the class *Register* which you wrote in a previous lab. Recall that the class modeled a simple sales register that processes a number of cash sales. Each sale consists of a number of different items. The register keeps track of the total amount of cash in the register from all sales, and keeps a running subtotal of the cost of items of the sale in progress. The items to be sold are instances of the class *RetailItem*.

Recall that the register provides the following functionality.

- *inRegister*: the total amount of cash in the register (a *double*).
- *saleTotal*: the (sub)total net cost of items in the current sale in progress (a *double*).
- *subtotal*: the net price for a specified number of units of a specified item (a *double*). First argument is a *RetailItem*, second argument is quantity.
- *reset*: clear the total amount of cash in the register and subtotal of the current sale to 0.0.
- *newSale*: end the sale in progress and start a new one, updating the cash in register with the current sale total, and clearing the sale total.
- *sell*: process the sale of one unit of a specified item.
- *sell*: process the sale of the specified number of units of a specified item. First argument is a *RetailItem*, second argument is quantity.

We make two changes to the specifications. First, a customer is given a discount depending on the net amount of the sale. Specifically, if the sale is more than \$10.00, the customer is given a 5% discount. If the sale is more than \$50.00, the customer is given a 10% discount. For simplicity, discounts are applied to the net sale total.

Second, we do not assume that there is sufficient quantity of an item to process a sales request. If there is not a sufficient quantity on hand to satisfy a customer's request, the customer is sold the number on hand. For instance, if the register is told to process the sale of 10 units of a given item and there are only three units on hand, only three items will be sold. The register client must be able to determine the actual number of units sold. So we add a method to the register:

- *quantitySold*: the number of units actually sold from the previous *sell* command; 0 if there was no previous *sell* command.

Specify the class *Register*:

- Copy the files from `~labCourse/Labs/Lab11/regsales11/` into a subdirectory of your *Java* directory named *regsales11*.
- Write specifications for the class *Register* in the package *regsales11*. You can do this by copying and modifying the class *regsales8.Register* that you wrote in a previous lab. Do the revision carefully!
- Include relevant preconditions and postconditions in your specification. The format that you should use is the word “*require:*” on a line by itself preceding preconditions in your doc comments, and the word “*ensure:*” on a line by itself preceding postconditions. For instance, here are examples from the class *RetailItem*:

```
/**
 * Increment the number of available units of this RetailItem by the
 * specified amount.
 *   require:
 *     count >= 0
 */
public void restock (int count) { ...

/**
 * The number of units of this RetailItem available for sale.
 *   ensure:
 *     this.onHand() >= 0
 */
public int onHand () { ...
```

Try to write preconditions and postconditions as legal Java boolean expressions whenever possible.

Create HTML specs for the class *Register* with *javadoc*:

Documentation convention is that class names, method names, *etc.* are written in “code” font. To produce source for *javadoc* that includes appropriate html tags, you can use the utility *prejavadoc*. This is a Perl script located in `~labCourse/utilities`. Input for the script (standard input) is a Java source file, and output (standard out) is the file with html tags added in doc comments. The script will put appropriate html tags around preconditions and postconditions, and will wrap `<code>...</code>` tags around words prefixed with *cx* and around lines prefixed with *cz* in doc comments.

For example, suppose that the method *restock* is written as

```
/**
 * Increment the number of available units of this cxRetailItem by the
 * specified amount.
 *   require:
 *     czcount >= 0
 */
public void restock (int count) { ...
```

and is in the file *RetailItem.java*. Then keying the command

```
~labCourse/utilities/prejavadoc < RetailItem.java > temp.java
```

or the command

```
perl ~labCourse/utilities/prejavadoc < RetailItem.java > temp.java
```

will create a file named *temp.java* that is the same as *RetailItem.java*, but includes some html tags in doc comments. In particular, the “require” clause will be properly set off, and the word “RetailItem” and the line “count >= 0” in the doc comment will be wrapped in `<code>...</code>` tags.

You can now replace the original file with the newly created one:

```
mv temp.java RetailItem.java
```

Implement and test the class *Register*:

- Carefully read the documentation of the class *regsales11.RetailItem* located *here*. Note in particular the preconditions specified for the methods you will invoke.
- Implement the class *regsales11.Register*. Make sure that you satisfy the preconditions of any classes you invoke.
- Be sure to use named constants as appropriate. Literals like “10.0” and “50.0” should appear only in the definition of a named constant.
- Test your implementation by running *regsales11.RegisterGUIStart*.

Post-lab:

As directed by your lab instructor, submit a listing of your *Register* class and the javadoc documentation of your *Register* class.