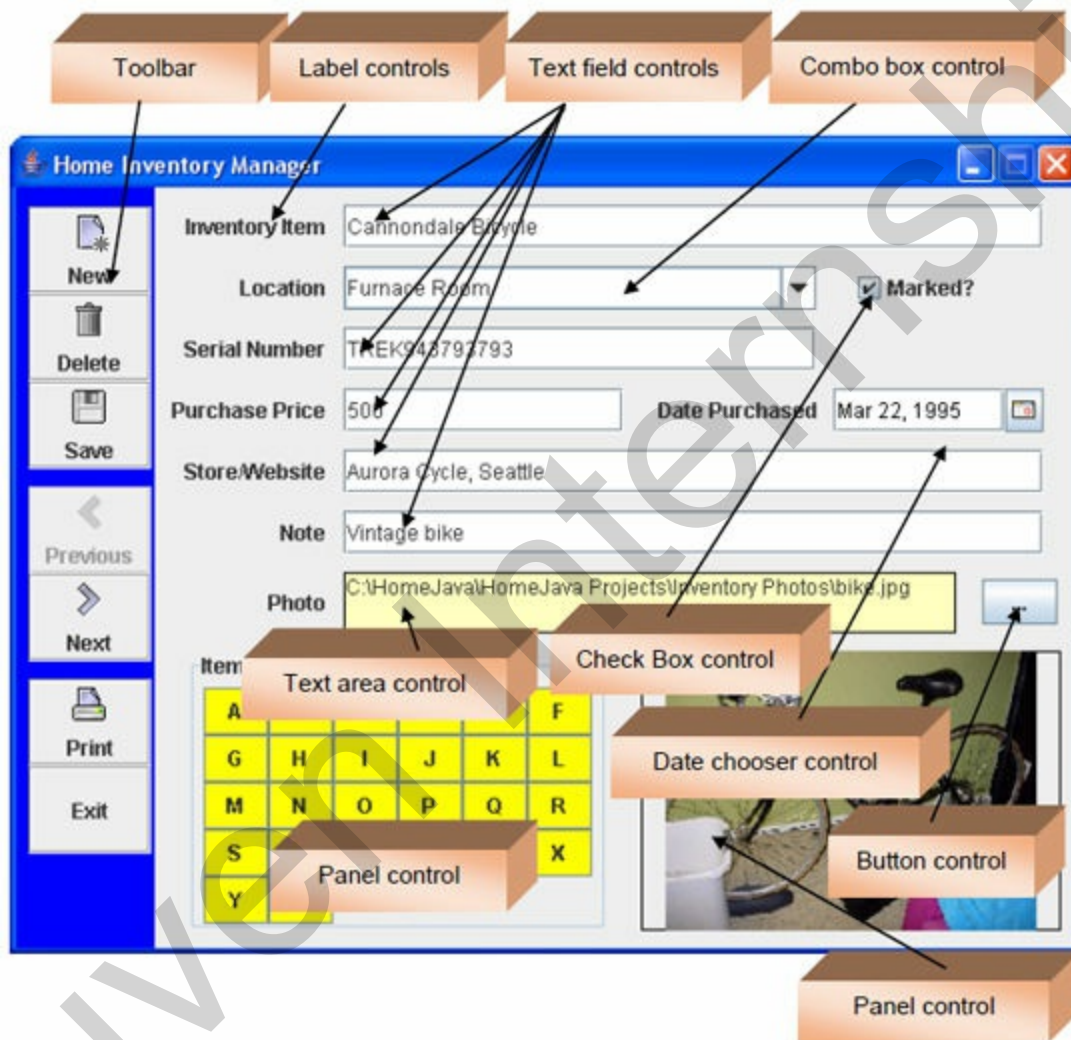


Home Inventory Manager Project Preview

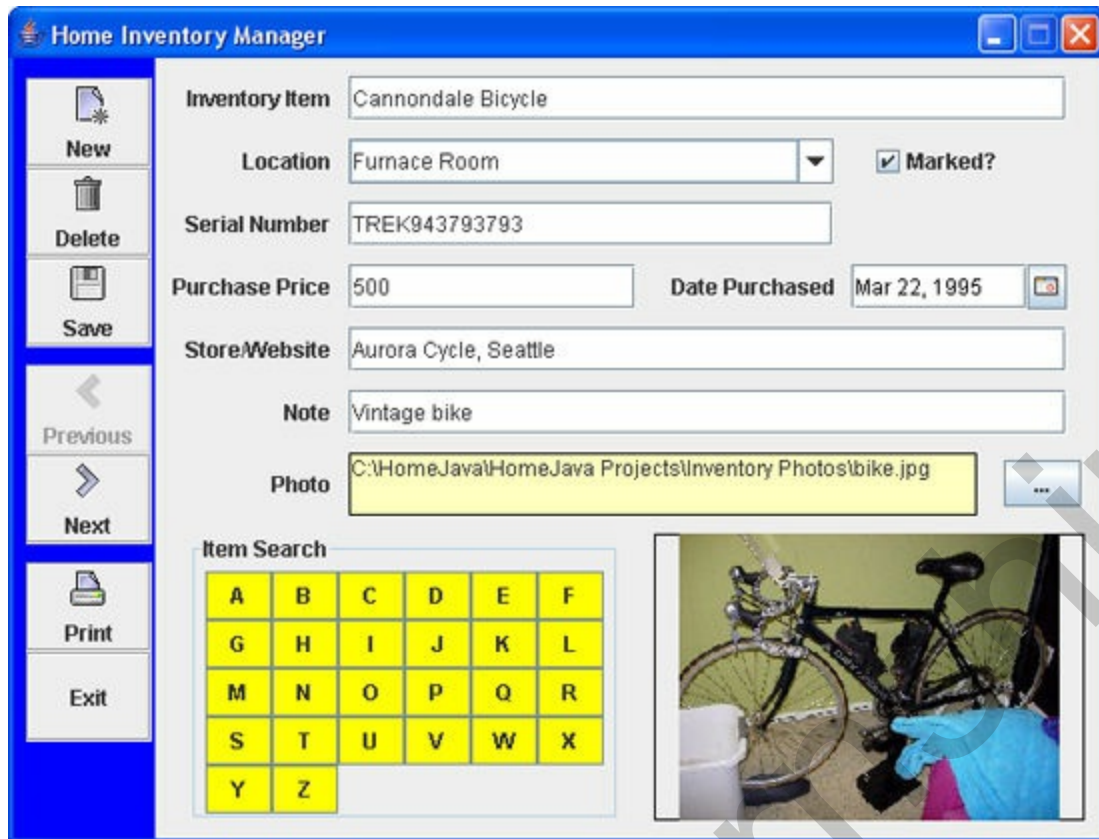
In this project, we will build a **home inventory manager** program. This program lets you keep a record of your belongings.

The finished project is saved as **HomeInventory** in the **\HomeJava\HomeJava Projects** project group. Start NetBeans (or your IDE). Open the specified project group. Make **HomeInventory** the main project. Run the project. You will see:



A toolbar control is used to add, delete and save items from the inventory. It is also used to navigate from one item to the next. The primary way to enter information about an inventory item is with several text field controls. A combo box is used to specify location, while a date chooser is used to select purchase date. A check box control indicates if an item is marked with identifying information. A button control (with an ellipsis) selects a photo to display in the panel control. A panel control holds 26 buttons for searching.

The program has a built-in sample inventory file – the first item in that file is displayed (items are listed alphabetically by **Inventory Item**):



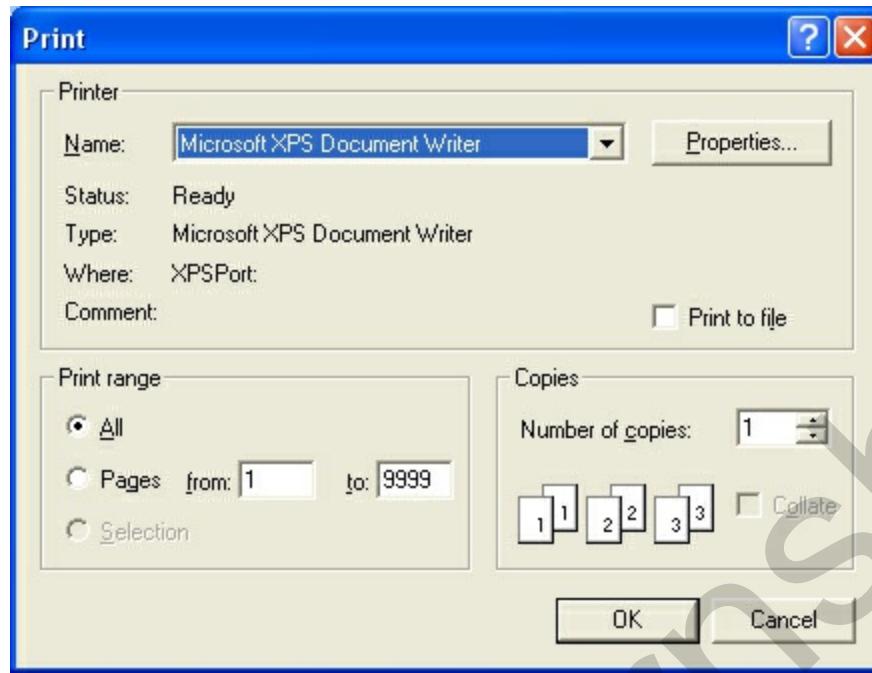
You will, of course, be able to replace the built-in file with your own belongings, but for now, let's [see](#) how the program works.

The idea of the program is to enter and/or view descriptive information about each item in your inventory. You can enter:

Inventory Item	A description of the item (required)
Location	Description of where item is located
Marked	Indicates if item is marked with some kind of identifying information (social security number, driver's license number, phone number)
Serial Number	Item serial number
Purchase Price	How much you paid for the item.
Date Purchased	When you purchased the item.
Store/Website	Where you purchased the item.
Note	Any additional information about the item.
Photo	View a stored JPEG photo of the item.

On the toolbar are two buttons marked **Previous** and **Next**. Use these to move from one item to the next. The sample file has 10 items to view. In the **Item Search** panel are 26 buttons, each with a letter of the alphabet. These are used to search through the inventory for items beginning with the clicked letter. Try searching the sample inventory, if you'd like.

Another nice feature of the project is the ability to get a printed record of your inventory. Click the toolbar button marked **Print** (don't worry, nothing will print). You will see:



This is the standard print dialog where you select printing options (including what printer to use). Click **Cancel**.

A primary task of the home inventory manager is to add, edit, save and delete inventory items. To add an item, you click the **Add** button in the toolbar. You then enter the necessary information and click the **Save** button. To edit an existing item, you first display the item to edit. Make the desired changes and click **Save**. To delete an item, you display the item, then click the **Delete** toolbar button. Let's try the editing features.

Navigate to one of the existing items in the sample file (use the **Previous** or **Next** buttons or try a search). I moved to **Toby**, my ever faithful dog:

Home Inventory Manager

Inventory Item: Toby

Location: Under the other desk ☒ Marked?

Serial Number: DOOFUS123

Purchase Price: 0.00 Date Purchased: Jun 6, 2001


Store/Website: Olympia SPCA

Note: Priceless

Photo: C:\HomeJava\HomeJava Projects\Inventory Photos\toby.jpg

Item Search:

A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R
S	T	U	V	W	X
Y	Z				



Navigation buttons: New, Delete, Save, Previous, Next, Print, Exit

We'll delete this item, then rebuild it to demonstrate how to enter information. Click the **Delete** button – choose **Yes** when asked if you really want Toby to go away. The display will show the next item in the inventory. Click the **New** button to start a new item.

The blank inventory screen appears as:

Home Inventory Manager

Inventory Item:

Location: ☐ Marked?

Serial Number:

Purchase Price: Date Purchased: Nov 28, 2008

Store/Website:

Note:

Photo:

Item Search:

A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R
S	T	U	V	W	X
Y	Z				

Navigation buttons: New, Delete, Save, Previous, Next, Print, Exit

At this point, you simply work your way down the form entering the desired information at the desired locations. When done, you click **Save** and the item is added to your inventory. We'll add Toby back to the file.

Under **Inventory Item**, type **Toby** and press <Enter>. This is the only required piece of information – all other entries are optional. For **Location**, click the drop-down arrow in the combo box. A list of choices is presented. Choose one of these items or type your own. If you type an entry that's not in the combo box, it will be added and saved for future items. Choose **Under the other desk** for Toby (he's always there). Put a check mark next to **Marked?** Make up a **Serial Number** for Toby – I used **DOOFUS123**. We got Toby for free, so his **Purchase Price** is **0.00**. We got Toby on **June 6, 2001**. Under **Date Purchased**, click the drop-down arrow. On the calendar that appears, navigate to this date and click it. Under **Store/Website**, type **Olympia SPCA** (he's a pound puppy) and under **Note**, type **Priceless**.

At this point, the form should look like this:

The screenshot shows a window titled "Home Inventory Manager" with a blue title bar. On the left is a vertical toolbar with buttons: "New" (plus icon), "Delete" (trash icon), "Save" (floppy disk icon), "Previous" (left arrow), "Next" (right arrow), "Print" (printer icon), and "Exit" (X icon). The main form area contains the following fields:

- Inventory Item:** Text box containing "Toby".
- Location:** Drop-down menu showing "Under the other desk".
- Marked?:** Check box that is checked.
- Serial Number:** Text box containing "DOOFUS123".
- Purchase Price:** Text box containing "0.00".
- Date Purchased:** Text box containing "Jun 6, 2001" with a calendar icon to its right.
- Store/Website:** Text box containing "Olympia SPCA".
- Note:** Text box containing "Priceless".
- Photo:** A yellow rectangular area with a blue button containing "..." to its right.

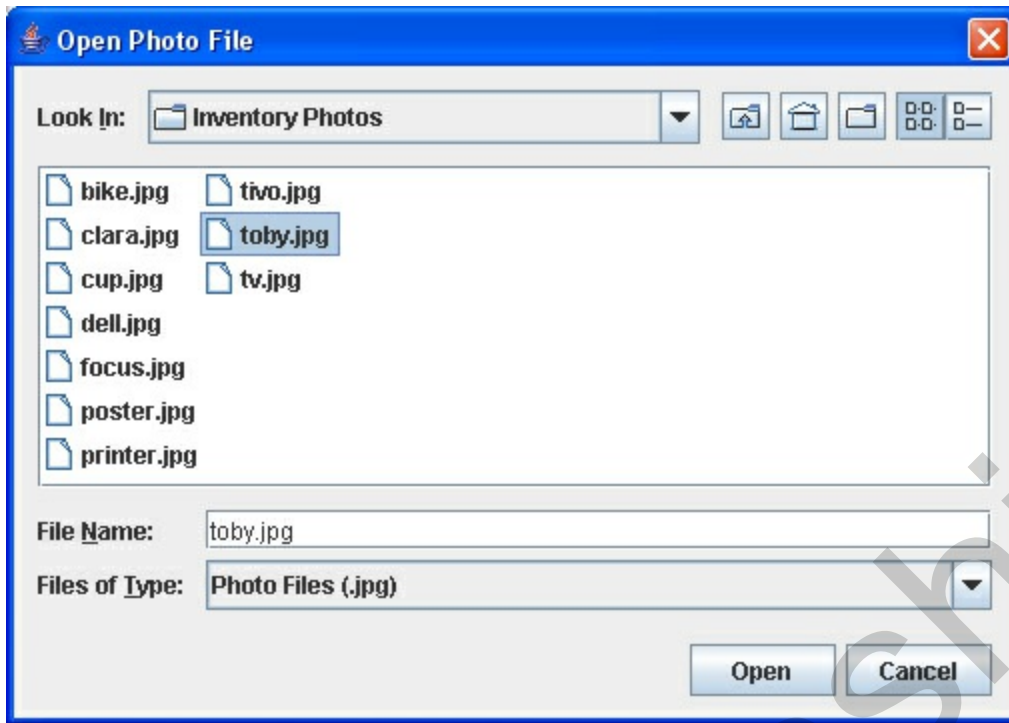
Below the "Photo" area is an "Item Search" section with a grid of yellow buttons labeled with letters A through Z. The grid is 5 rows by 6 columns, with the last row containing only "Y" and "Z".

A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R
S	T	U	V	W	X
Y	Z				

To the right of the "Item Search" grid is a large, empty rectangular area.

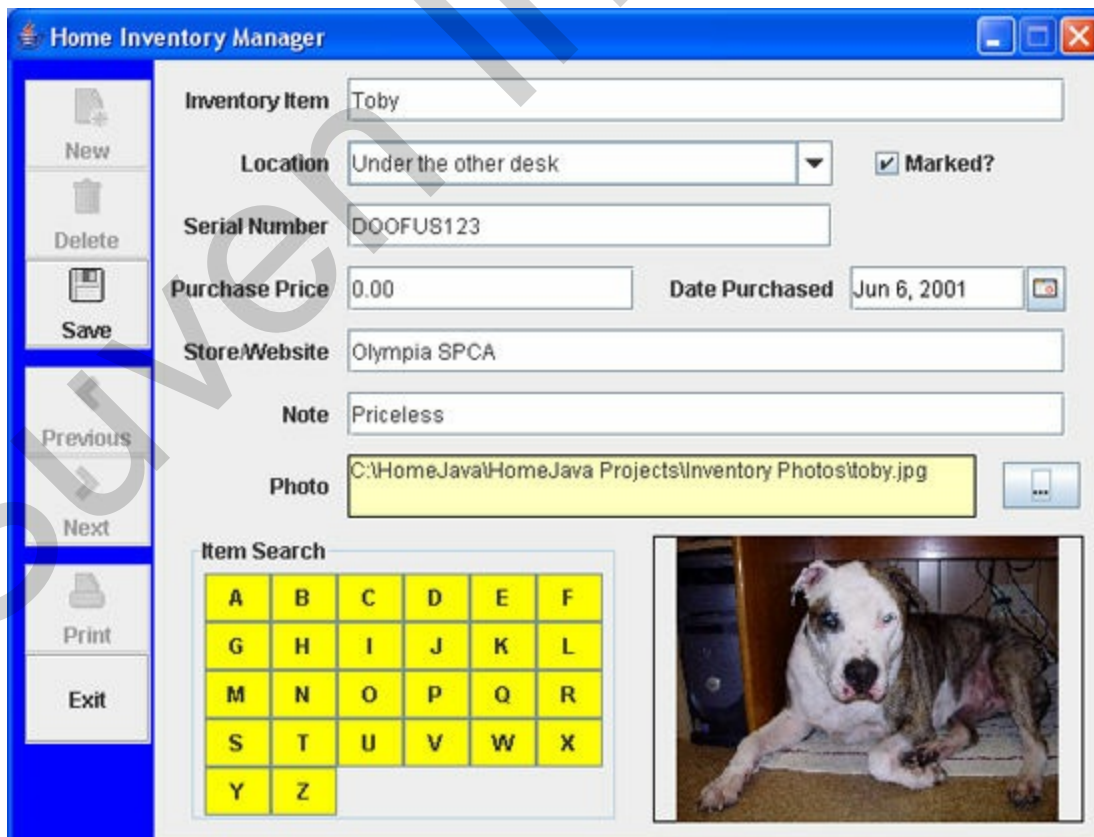
The last step is adding a photo.

Click the button with the ellipsis (...) next to the **Photo** label area. An open file dialog box will appear:



The photo can be any JPEG file (what a digital camera uses). You simply navigate to a photo location and click **Open**. The samples for these notes are in the **\HomeJava\HomeJava Projects\Inventory Photos** folder. Move to that folder and select **toby.jpg** as shown. Click **Open** and the photo will appear.

The final **Toby** inventory item page looks like this:



Notice the photo and the file name listed under **Photo**. At this point, click **Save** and Toby is back in the list (properly sorted alphabetically).

That's the idea of the program. Fill in an entry page for each item in your inventory and click **Save**. Click **Exit** on the toolbar when done. Upon exiting the program, all your inventory items are saved to a file (the built-in file currently holding the sample entries). This same file is automatically opened when you rerun the program, so your items are always available for additions, changes and deletions.

We will now build this program in several stages. We discuss **frame design**. We discuss the controls used to build the frame and establish initial properties. We see how to add a toolbar to the project. And, we address **code design** in detail. We introduce object-oriented programming (OOP) concepts to store the inventory data. We discuss how to read and write the inventory file, how to perform the various editing features, how to load a photo file, how to create the buttons used in the search function, and how to print out the inventory.

Home Inventory Manager Frame Design

We begin building the **Home Inventory Project**. Let's build the frame. Start a new project in your Java project group – name it **HomeInventory**. Once started, we suggest you immediately save the project with the name you chose. This sets up the folder and file structure needed for your project. Build the basic frame with these properties:

Home Inventory Frame:

title	Home Inventory Manager
resizable	false

The code is:

```
/*
 * HomeInventory.java
 */
package homeinventory;
import javax.swing.*.*;
import java.awt.*.*;
import java.awt.event.*.*;

public class HomeInventory extends JFrame
{
    public static void main(String args[])
    {
        // create frame
        new HomeInventory().show();
    }

    public HomeInventory()
    {
        // frame constructor
        setTitle("Home Inventory Manager");
        setResizable(false);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent evt)
```



```

    {
        exitForm(evt);
    }
});

getContentPane().setLayout(new GridBagLayout());
GridBagConstraints gridConstraints;

pack();
Dimension screenSize =
Toolkit.getDefaultToolkit().getScreenSize();
    setBounds((int) (0.5 * (screenSize.width - getWidth())), (int) (0.5 * (screenSize.height
- getHeight())), getWidth(), getHeight());
}

private void exitForm(WindowEvent evt)
{
    System.exit(0);
}
}

```

This code builds the frame, sets up the layout manager and includes code to exit the application. Run the code to make sure the frame (at least, what there is of it at this point) appears and is centered on the screen:



Let's populate our frame with other controls. All code for creating the frame and placing controls (except declarations) goes in the **HomeInventory** constructor.

There are lots of controls in this project. The **GridBagLayout** for the frame is::

	gridx = 0	gridx = 1	gridx = 2	gridx = 3	gridx = 4	gridx = 5	gridx = 6
gridy = 0	inventoryToolBar	itemLabel	itemTextField				
gridy = 1		locationLabel	locationTextField		markedCheckBox		
gridy = 2		serialLabel	serialTextField				
gridy = 3		priceLabel	priceTextField	dateLabel	dateDateChooser		
gridy = 4		storeLabel	storeTextField				
gridy = 5		noteLabel	noteTextField				
gridy = 6		photoLabel	photoTextArea				photoButton
gridy = 7		searchPanel			photoPanel		

The label controls are used for titling. **photoTextArea** holds the plot file name. The text fields are used to input item information. The combo box control (**locationCombo**) is used to select location information. The check box (**markedCheckBox**) indicates if an item is marked. The date chooser (**dateDateChooser**) selects purchase date. The button (**photoButton**) is used to select the photo file. The photo is displayed in **photoPanel**. The other panel (**searchPanel**) will hold buttons used for searching. The tool bar (**inventoryToolbar**) is used to edit items and navigate from one item to the next.

As noted, there are lots of controls. We will discuss adding controls in a few steps. First, we discuss the toolbar. Then, we cover the controls used to input inventory information. Lastly, we add the **searchPanel** (and associated button controls) and the **photoPanel**.

Frame Design – Toolbar

The toolbar (**JToolbar**) is used to edit the inventory items and navigate through them. It is also used to print the items and exit the program. Let's preview what it should look like:



The toolbar will have seven buttons: one to create a **new** item (**newButton**), one to **delete** an item (**deleteButton**), one to **save** an item (**saveButton**), one to view the **previous** item (**previousButton**) one to view the **next** item (**nextButton**), one to **print** the inventory (**printButton**) and one to **exit** the program (**exitButton**). Separators are used at the top, after the **save** button and after the **next** button.

All but the last button has an image. The images used are to be goggled and downloaded by self. Copy these images to your project folder. The six images *with their names* are:



The control properties associated with the toolbar:

inventoryToolbar:

floatable	false
background	Blue
orientation	Vertical
gridx	0
gridy	0

gridheight	8
fill	Vertical

newbutton:

image	new.gif
text	New
size	70, 50
toolTipText	Add New Item
horizontalTextPosition	Center
verticalTextPosition	Bottom

deletebutton:

image	delete.gif
text	Delete
size	70, 50
toolTipText	Delete Current Item
horizontalTextPosition	Center
verticalTextPosition	Bottom

savebutton:

image	save.gif
text	Save
size	70, 50
toolTipText	Save Current Item
horizontalTextPosition	Center
verticalTextPosition	Bottom

previousbutton:

image	previous.gif
text	Previous
size	70, 50
toolTipText	Display Previous Item
horizontalTextPosition	Center
verticalTextPosition	Bottom

nextbutton:

image	next.gif
text	Next

size	70, 50
toolTipText	Display Next Item
horizontalTextPosition	Center
verticalTextPosition	Bottom

printbutton:

image	print.gif
text	Print
size	70, 50
toolTipText	Print Inventory List
horizontalTextPosition	Center
verticalTextPosition	Bottom

exitButton:

text	Exit
size	70, 50
toolTipText	Exit Program

Declare the controls as class level objects:

// Toolbar

```
JToolBar inventoryToolBar = new JToolBar();
JButton newButton = new JButton(new ImageIcon("new.gif"));
JButton deleteButton = new JButton(new ImageIcon("delete.gif"));
JButton saveButton = new JButton(new ImageIcon("save.gif"));
JButton previousButton = new JButton(new ImageIcon("previous.gif"));
JButton nextButton = new JButton(new ImageIcon("next.gif"));
JButton printButton = new JButton(new ImageIcon("print.gif"));
JButton exitButton = new JButton();
```

Add the controls (and separators) to the frame/toolbar using this code in the frame constructor:

```
inventoryToolBar.setFloatable(false);
inventoryToolBar.setBackground(Color.BLUE);
inventoryToolBar.setOrientation(SwingConstants.VERTICAL);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 0;
gridConstraints.gridy = 0;
```



```
gridConstraints.gridheight = 8;  
gridConstraints.fill = GridBagConstraints.VERTICAL;  
getContentPane().add(inventoryToolBar, gridConstraints);  
  
inventoryToolBar.addSeparator();
```

```
Dimension bSize = new Dimension(70, 50);  
newButton.setText("New");  
sizeButton(newButton, bSize);  
newButton.setToolTipText("Add New Item");  
newButton.setHorizontalTextPosition(SwingConstants.CENTER);  
newButton.setVerticalTextPosition(SwingConstants.BOTTOM);  
inventoryToolBar.add(newButton);  
newButton.addActionListener(new ActionListener()  
{  
    public void actionPerformed(ActionEvent e)  
    {  
        newButtonActionPerformed(e);  
    }  
});  
  
deleteButton.setText("Delete");  
sizeButton(deleteButton, bSize);  
deleteButton.setToolTipText("Delete Current Item");  
deleteButton.setHorizontalTextPosition(SwingConstants.CENTER);  
deleteButton.setVerticalTextPosition(SwingConstants.BOTTOM);  
inventoryToolBar.add(deleteButton);  
deleteButton.addActionListener(new ActionListener()  
{  
    public void actionPerformed(ActionEvent e)  
    {  
        deleteButtonActionPerformed(e);  
    }  
});
```

```
saveButton.setText("Save");  
sizeButton(saveButton, bSize);
```

```
saveButton.setToolTipText("Save Current Item");
saveButton.setHorizontalTextPosition(SwingConstants.CENTER);
saveButton.setVerticalTextPosition(SwingConstants.BOTTOM);
inventoryToolBar.add(saveButton);
saveButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        saveButtonActionPerformed(e);
    }
});

inventoryToolBar.addSeparator();

previousButton.setText("Previous");
sizeButton(previousButton, bSize);
previousButton.setToolTipText("Display Previous Item");
previousButton.setHorizontalTextPosition(SwingConstants.CENTER);
previousButton.setVerticalTextPosition(SwingConstants.BOTTOM);
inventoryToolBar.add(previousButton);
previousButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        previousButtonActionPerformed(e);
    }
});

nextButton.setText("Next");
sizeButton(nextButton, bSize);
nextButton.setToolTipText("Display Next Item");
nextButton.setHorizontalTextPosition(SwingConstants.CENTER);
nextButton.setVerticalTextPosition(SwingConstants.BOTTOM);
inventoryToolBar.add(nextButton);
nextButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
```

```

    {
        nextButtonActionPerformed(e);
    }
});

```

```

inventoryToolBar.addSeparator();

```

```

printButton.setText("Print");
sizeButton(printButton, bSize);
printButton.setToolTipText("Print Inventory List");
printButton.setHorizontalTextPosition(SwingConstants.CENTER);
printButton.setVerticalTextPosition(SwingConstants.BOTTOM);
inventoryToolBar.add(printButton);
printButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        printButtonActionPerformed(e);
    }
});

```

```

exitButton.setText("Exit");
sizeButton(exitButton, bSize);
exitButton.setToolTipText("Exit Program");
inventoryToolBar.add(exitButton);
exitButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        exitButtonActionPerformed(e);
    }
});

```

Note each toolbar button is the same size (70 x 50). We defined a variable (**bSize**) to represent the size and used a general method (**sizeButton**) to set the size.. Add this method to your project:

```

private void sizeButton(JButton b, Dimension d)

```

```
{  
    b.setPreferredSize(d);  
    b.setMinimumSize(d);  
    b.setMaximumSize(d);  
}
```

Each button has an **ActionPerformed** method. Add these empty methods:

```
private void newButtonActionPerformed(ActionEvent e)  
{  
}  
  
private void deleteButtonActionPerformed(ActionEvent e)  
{  
}  
  
private void saveButtonActionPerformed(ActionEvent e)  
{  
}  
  
private void previousButtonActionPerformed(ActionEvent e)  
{  
}  
  
private void nextButtonActionPerformed(ActionEvent e)  
{  
}  
  
private void printButtonActionPerformed(ActionEvent e)  
{  
}  
  
private void exitButtonActionPerformed(ActionEvent e)  
{  
}
```

Save and run the project. You should see the completed toolbar on the left side of the frame:



Frame Design – Entry Controls

We show the frame **GridBagLayout** again to allow placement of all the controls used to input information about items:

	gridx = 0	gridx = 1	gridx = 2	gridx = 3	gridx = 4	gridx = 5	gridx = 6
gridy = 0	inventoryToolBar	itemLabel	itemTextField				
gridy = 1		locationLabel	locationTextField		markedCheckBox		
gridy = 2		serialLabel	serialTextField				
gridy = 3		priceLabel	priceTextField	dateLabel	dateDateChooser		
gridy = 4		storeLabel	storeTextField				
gridy = 5		noteLabel	noteTextField				
gridy = 6		photoLabel	photoTextArea			photoButton	
gridy = 7		searchPanel			photoPanel		

We will now add everything but the two panels at the bottom of the grid. There are many controls and many properties. We will add them in stages to keep things manageable. First the controls in the first three grid rows.

The control properties are:

itemLabel:

text Inventory Item
gridx 1
gridy 0
insets 10, 10, 0, 10
anchor EAST

itemTextField:

size 400, 25
gridx 2
gridy 0
gridwidth 5
insets 10, 0, 0, 10
anchor WEST

locationLabel:

text Location
gridx 1
gridy 1

insets	10, 10, 0, 10
anchor	EAST

locationComboBox:

size	270, 25
font	Arial, Plain, Size 12
editable	true
background	White
gridx	2
gridy	1
gridwidth	3
insets	10, 0, 0, 10
anchor	WEST

markedCheckBox:

text	Marked?
gridx	5
gridy	1
insets	10, 10, 0, 0
anchor	WEST

serialLabel:

text	Serial Number
gridx	1
gridy	2
insets	10, 10, 0, 10
anchor	EAST

serialTextField:

size	270, 25
gridx	2
gridy	2
gridwidth	3
insets	10, 0, 0, 10
anchor	WEST

Declare the controls as class level objects:

// Frame

```
JLabel itemLabel = new JLabel();  
JTextField itemTextField = new JTextField();  
JLabel locationLabel = new JLabel();  
JComboBox locationComboBox = new JComboBox();  
JCheckBox markedCheckBox = new JCheckBox();  
JLabel serialLabel = new JLabel();  
JTextField serialTextField = new JTextField();
```

Add the controls to the frame using:

```
itemLabel.setText("Inventory Item");  
gridConstraints = new GridBagConstraints();  
gridConstraints.gridx = 1;  
gridConstraints.gridy = 0;  
gridConstraints.insets = new Insets(10, 10, 0, 10);  
gridConstraints.anchor = GridBagConstraints.EAST;  
getContentPane().add(itemLabel, gridConstraints);  
  
itemTextField.setPreferredSize(new Dimension(400, 25));  
gridConstraints = new GridBagConstraints();  
gridConstraints.gridx = 2;  
gridConstraints.gridy = 0;  
gridConstraints.gridwidth = 5;  
gridConstraints.insets = new Insets(10, 0, 0, 10);  
gridConstraints.anchor = GridBagConstraints.WEST;  
getContentPane().add(itemTextField, gridConstraints);  
  
locationLabel.setText("Location");  
gridConstraints = new GridBagConstraints();  
gridConstraints.gridx = 1;  
gridConstraints.gridy = 1;  
gridConstraints.insets = new Insets(10, 10, 0, 10);  
gridConstraints.anchor = GridBagConstraints.EAST;  
getContentPane().add(locationLabel, gridConstraints);  
locationComboBox.setPreferredSize(new Dimension(270, 25));  
locationComboBox.setFont(new Font("Arial", Font.PLAIN, 12));
```

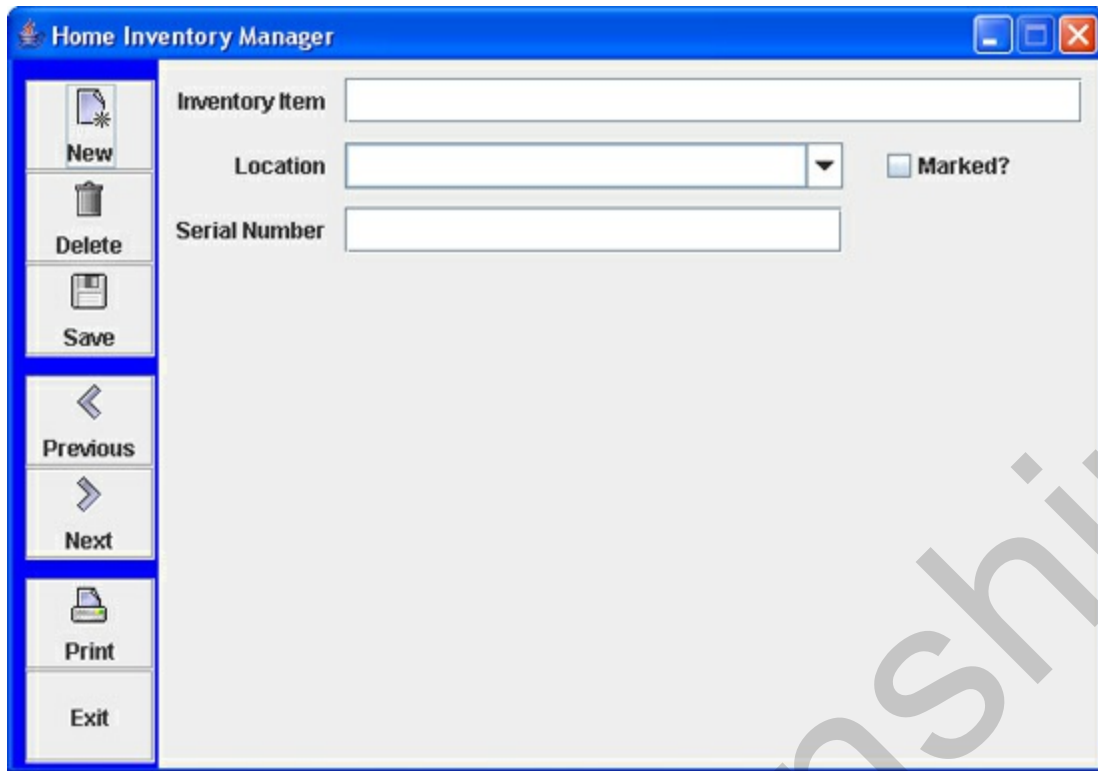
```
locationComboBox.setEditable(true);
locationComboBox.setBackground(Color.WHITE);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 2;
gridConstraints.gridy = 1;
gridConstraints.gridwidth = 3;
gridConstraints.insets = new Insets(10, 0, 0, 10);
gridConstraints.anchor = GridBagConstraints.WEST;
getContentPane().add(locationComboBox, gridConstraints);

markedCheckBox.setText("Marked?");
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 5;
gridConstraints.gridy = 1;
gridConstraints.insets = new Insets(10, 10, 0, 0);
gridConstraints.anchor = GridBagConstraints.WEST;
getContentPane().add(markedCheckBox, gridConstraints);

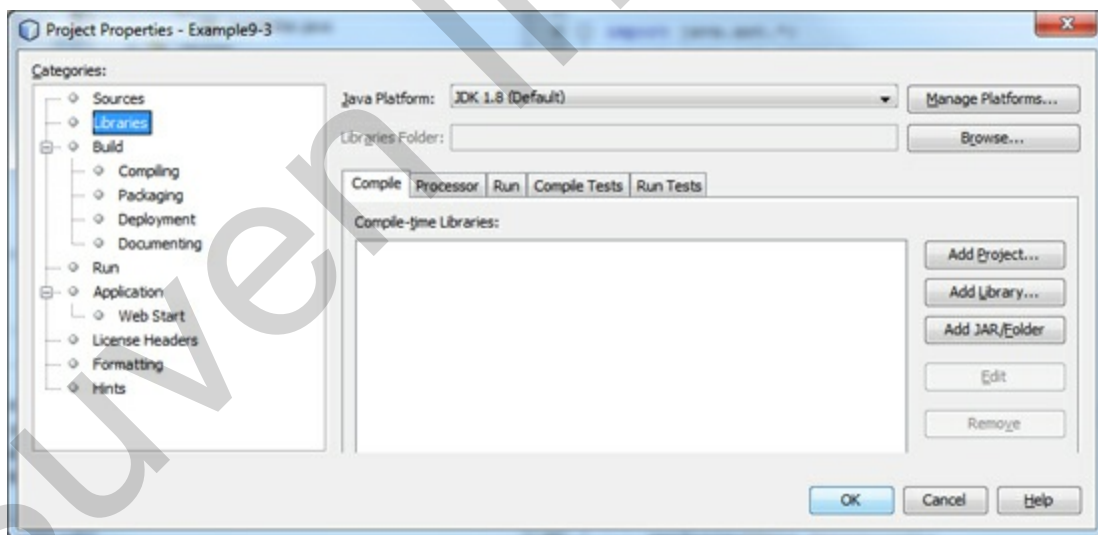
serialLabel.setText("Serial Number");
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 1;
gridConstraints.gridy = 2;
gridConstraints.insets = new Insets(10, 10, 0, 10);
gridConstraints.anchor = GridBagConstraints.EAST;
getContentPane().add(serialLabel, gridConstraints);

serialTextField.setPreferredSize(new Dimension(270, 25));
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 2;
gridConstraints.gridy = 2;
gridConstraints.gridwidth = 3;
gridConstraints.insets = new Insets(10, 0, 0, 10);
gridConstraints.anchor = GridBagConstraints.WEST;
getContentPane().add(serialTextField, gridConstraints);
```

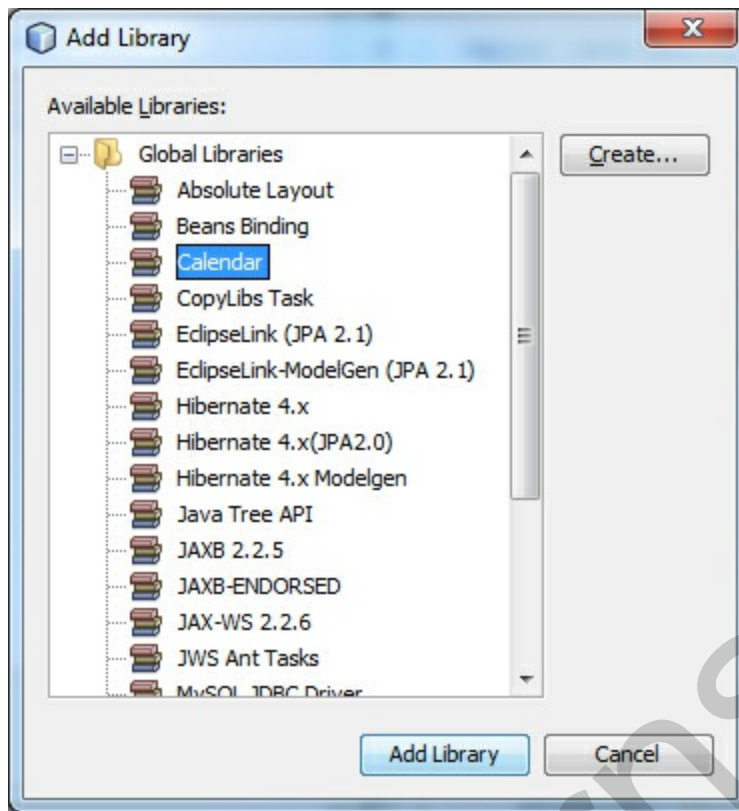
Run to see the newly added controls:



Let's add the remaining input controls. For the date chooser control, we have to add the **jcalendar-1.3.2.jar** library to our NetBeans / Eclipse IDE. We need to add this library to our project. Make sure **HomeInventory** is the active project. In the file view area, right-click the project name (**HomeInventory**) and click **Properties**. In the properties window, choose the **Libraries** category:



Click **Add Library** to see



Choose **Calendar**, then click **Add Library**. Click **OK** when returned to the **Properties** window. The calendar tools can now be used in the weight monitor project with the addition of these **import** statements:

```
import com.toedter.calendar.*;
import java.beans.*;
```

Add these to the code window.

The control properties are:

priceLabel:

text	Purchase Price
gridx	1
gridy	3
insets	10, 10, 0, 10
anchor	EAST

priceTextField:

size	160, 25
gridx	2
gridy	3
gridwidth	1

insets	10, 0, 0, 10
anchor	WEST

dateLabel:

text	Date Purchased
gridx	4
gridy	3
insets	10, 10, 0, 0
anchor	EAST

dateDateChooser:

size	120, 25
gridx	5
gridy	3
gridwidth	2
insets	10, 0, 0, 10
anchor	WEST

storeLabel:

text	Store/Website
gridx	1
gridy	4
insets	10, 10, 0, 10
anchor	EAST

storeTextField:

size	400, 25
gridx	2
gridy	4
gridwidth	5
insets	10, 0, 0, 10
anchor	WEST

noteLabel:

text	Note
gridx	1
gridy	5
insets	10, 10, 0, 10

anchor EAST

noteTextField:

size 400, 25
gridx 2
gridy 5
gridwidth 5
insets 10, 0, 0, 10
anchor WEST

photoLabel:

text Photo
gridx 1
gridy 6
insets 10, 10, 0, 10
anchor EAST

photoTextArea:

size 350, 35
font Arial, Plain, Size 12
editable false
lineWrap true
wrapStyleWord true
background Color(255, 255, 192)
border Black line
gridx 2
gridy 6
gridwidth 4
insets 10, 0, 0, 10
anchor WEST

photoButton:

text ...
gridx 6
gridy 6
insets 10, 0, 0, 10
anchor WEST

These controls are declared using:

```
JLabel priceLabel = new JLabel();  
JTextField priceTextField = new JTextField();  
JLabel dateLabel = new JLabel();  
JDateChooser dateDateChooser = new JDateChooser();  
JLabel storeLabel = new JLabel();  
JTextField storeTextField = new JTextField();  
JLabel noteLabel = new JLabel();  
JTextField noteTextField = new JTextField();  
JLabel photoLabel = new JLabel();  
JTextArea photoTextArea = new JTextArea();  
JButton photoButton = new JButton();
```

The controls are added to the frame using:

```
priceLabel.setText("Purchase Price");  
gridConstraints = new GridBagConstraints();  
gridConstraints.gridx = 1;  
gridConstraints.gridy = 3;  
gridConstraints.insets = new Insets(10, 10, 0, 10);  
gridConstraints.anchor = GridBagConstraints.EAST;  
getContentPane().add(priceLabel, gridConstraints);  
  
priceTextField.setPreferredSize(new Dimension(160, 25));  
gridConstraints = new GridBagConstraints();  
gridConstraints.gridx = 2;  
gridConstraints.gridy = 3;  
gridConstraints.gridwidth = 2;  
gridConstraints.insets = new Insets(10, 0, 0, 10);  
gridConstraints.anchor = GridBagConstraints.WEST;  
getContentPane().add(priceTextField, gridConstraints);  
  
dateLabel.setText("Date Purchased");  
gridConstraints = new GridBagConstraints();  
gridConstraints.gridx = 4;  
gridConstraints.gridy = 3;
```

```
gridConstraints.insets = new Insets(10, 10, 0, 0);  
gridConstraints.anchor = GridBagConstraints.WEST;  
getContentPane().add(dateLabel, gridConstraints);
```

```
dateDateChooser.setPreferredSize(new Dimension(120, 25));  
gridConstraints = new GridBagConstraints();  
gridConstraints.gridx = 5;  
gridConstraints.gridy = 3;  
gridConstraints.gridwidth = 2;  
gridConstraints.insets = new Insets(10, 0, 0, 10);  
gridConstraints.anchor = GridBagConstraints.WEST;  
getContentPane().add(dateDateChooser, gridConstraints);
```

```
storeLabel.setText("Store/Website");  
gridConstraints = new GridBagConstraints();  
gridConstraints.gridx = 1;  
gridConstraints.gridy = 4;  
gridConstraints.insets = new Insets(10, 10, 0, 10);  
gridConstraints.anchor = GridBagConstraints.EAST;  
getContentPane().add(storeLabel, gridConstraints);
```

```
storeTextField.setPreferredSize(new Dimension(400, 25));  
gridConstraints = new GridBagConstraints();  
gridConstraints.gridx = 2;  
gridConstraints.gridy = 4;  
gridConstraints.gridwidth = 5;  
gridConstraints.insets = new Insets(10, 0, 0, 10);  
gridConstraints.anchor = GridBagConstraints.WEST;  
getContentPane().add(storeTextField, gridConstraints);
```

```
noteLabel.setText("Note");  
gridConstraints = new GridBagConstraints();  
gridConstraints.gridx = 1;  
gridConstraints.gridy = 5;  
gridConstraints.insets = new Insets(10, 10, 0, 10);  
gridConstraints.anchor = GridBagConstraints.EAST;  
getContentPane().add(noteLabel, gridConstraints);
```



```
noteTextField.setPreferredSize(new Dimension(400, 25));  
gridConstraints = new GridBagConstraints();  
gridConstraints.gridx = 2;  
gridConstraints.gridy = 5;  
gridConstraints.gridwidth = 5;  
gridConstraints.insets = new Insets(10, 0, 0, 10);  
gridConstraints.anchor = GridBagConstraints.WEST;  
getContentPane().add(noteTextField, gridConstraints);
```

```
photoLabel.setText("Photo");  
gridConstraints = new GridBagConstraints();  
gridConstraints.gridx = 1;  
gridConstraints.gridy = 6;  
gridConstraints.insets = new Insets(10, 10, 0, 10);  
gridConstraints.anchor = GridBagConstraints.EAST;  
getContentPane().add(photoLabel, gridConstraints);
```

```
photoTextArea.setPreferredSize(new Dimension(350, 35));  
photoTextArea.setFont(new Font("Arial", Font.PLAIN, 12));  
photoTextArea.setEditable(false);  
photoTextArea.setLineWrap(true);  
photoTextArea.setWrapStyleWord(true);  
photoTextArea.setBackground(new Color(255, 255, 192));  
photoTextArea.setBorder(BorderFactory.createLineBorder(Color.BLACK));  
gridConstraints = new GridBagConstraints();  
gridConstraints.gridx = 2;  
gridConstraints.gridy = 6;  
gridConstraints.gridwidth = 4;  
gridConstraints.insets = new Insets(10, 0, 0, 10);  
gridConstraints.anchor = GridBagConstraints.WEST;  
getContentPane().add(photoTextArea, gridConstraints);
```

```
photoButton.setText("...");  
gridConstraints = new GridBagConstraints();  
gridConstraints.gridx = 6;  
gridConstraints.gridy = 6;  
gridConstraints.insets = new Insets(10, 0, 0, 10);
```

```

gridConstraints.anchor = GridBagConstraints.WEST;
getContentPane().add(photoButton, gridConstraints);
photoButton.addActionListener(new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {
        photoButtonActionPerformed(e);
    }
});

```

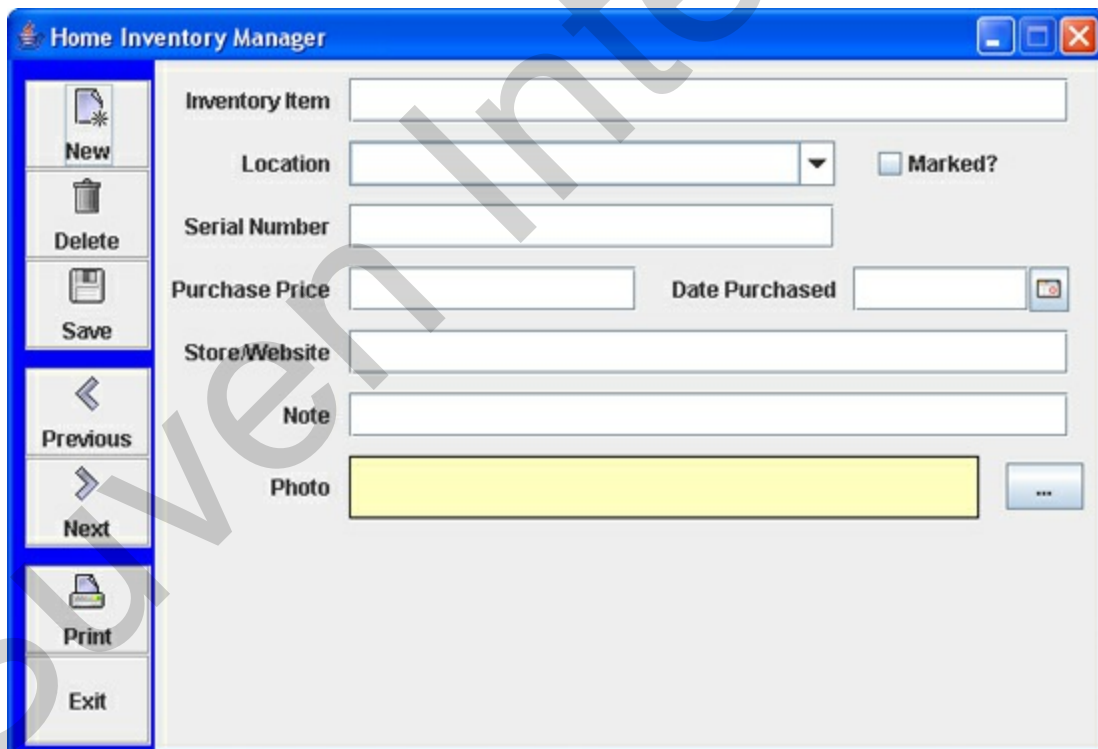
The button control has an **ActionPerformed** method. Add the empty method

```

private void photoButtonActionPerformed(ActionEvent e)
{
}

```

Run to see more controls:



Frame Design – Search Panel

As an inventory list grows, you will want the capability to search for particular items. In this project, we use 26 small button controls (**searchButton** array) in the **searchPanel** control. Each of these buttons will have a letter of the alphabet. When a letter is clicked, the first item in the inventory beginning with that letter (if there is such an item) is displayed on the frame.

The **searchPanel GridBagLayout** is (shown is the letter of the button to be displayed):

	gridx = 0	gridx = 1	gridx = 2	gridx = 3	gridx = 4	gridx = 5
gridy = 0	A	B	C	D	E	F
gridy = 1	G	H	I	J	K	L
gridy = 2	M	N	O	P	Q	R
gridy = 3	S	T	U	V	W	X
gridy = 4	Y	Z				

The panel and button properties are:

searchPanel:

size	240, 160
title	Item Search
gridx	1
gridy	7
gridwidth	3
insets	10, 0, 10, 0
anchor	CENTER

searchButton:

font	Arial, Bold, Size 12
margin	-10, -10, -10, -10
size	37, 27
background	Yellow

We only list generic button properties. The grid above shows the corresponding letter and location. Using negative margins allows room for the text. The button size (37 x 27) was found by trial and error.

Declare the panel and buttons using:

```
JPanel searchPanel = new JPanel();
```

```
JButton[] searchButton = new JButton[26];
```

Add them to the panel using:

```
searchPanel.setPreferredSize(new Dimension(240, 160));  
searchPanel.setBorder(BorderFactory.createTitledBorder("Item Search"));  
searchPanel.setLayout(new GridBagLayout());  
gridConstraints = new GridBagConstraints();  
gridConstraints.gridx = 1;  
gridConstraints.gridy = 7;  
gridConstraints.gridwidth = 3;  
gridConstraints.insets = new Insets(10, 0, 10, 0);  
gridConstraints.anchor = GridBagConstraints.CENTER;  
getContentPane().add(searchPanel, gridConstraints);
```

```
int x = 0, y = 0;  
// create and position 26 buttons  
for (int i = 0; i < 26; i++)  
{  
    // create new button  
    searchButton[i] = new JButton();  
    // set text property  
    searchButton[i].setText(String.valueOf((char) (65 + i)));  
    searchButton[i].setFont(new Font("Arial", Font.BOLD, 12));  
    searchButton[i].setMargin(new Insets(-10, -10, -10, -10));  
    sizeButton(searchButton[i], new Dimension(37, 27));  
    searchButton[i].setBackground(Color.YELLOW);  
    gridConstraints = new GridBagConstraints();  
    gridConstraints.gridx = x;  
    gridConstraints.gridy = y;  
    searchPanel.add(searchButton[i], gridConstraints);  
    // add method  
    searchButton[i].addActionListener(new ActionListener ()  
    {  
        public void actionPerformed(ActionEvent e)  
        {  
            searchButtonActionPerformed(e);
```

```

    }
});
x++;
// six buttons per row
if (x % 6 == 0)
{
    x = 0;
    y++;
}
}

```

Note the buttons are sized using the previously added **sizeButton** method.

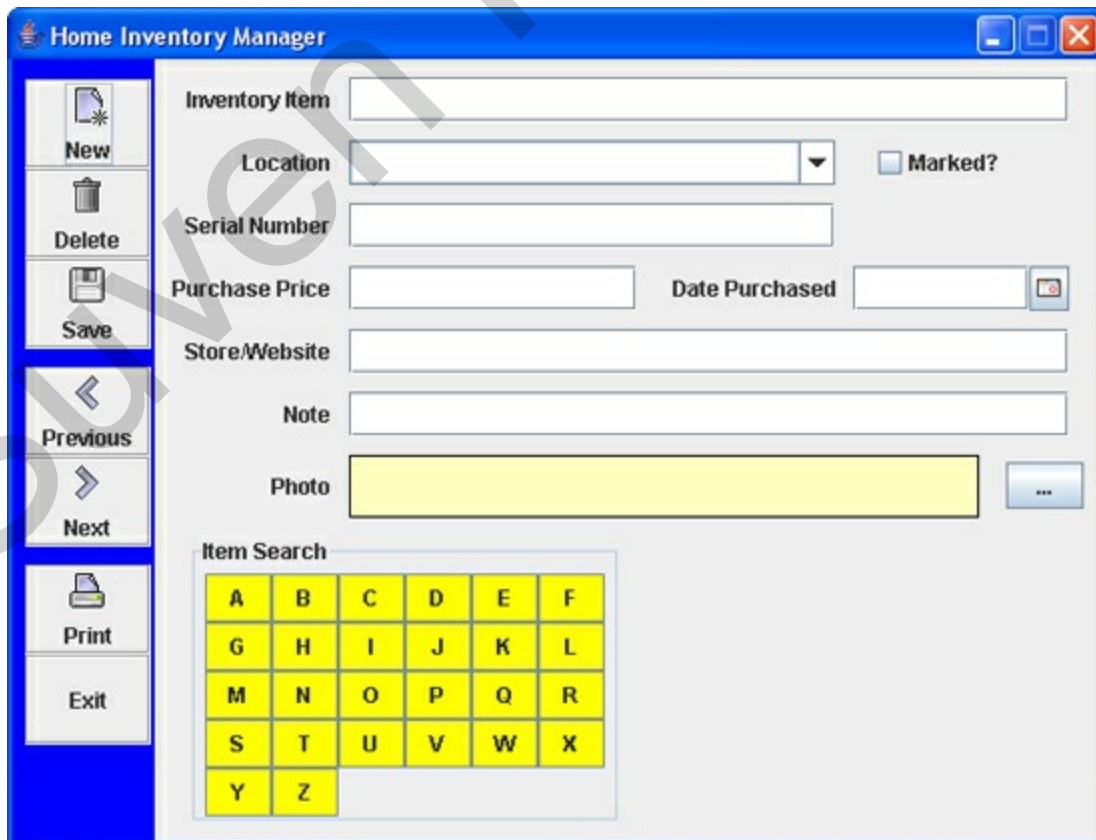
Each button has the same **ActionPerformed** method. Add this empty framework:

```

private void searchButtonActionPerformed(ActionEvent e)
{
}

```

Save and run the project. The search buttons should now appear on the frame in the search panel control:



Frame Design – Photo Panel

We complete the frame design by adding the panel (**photoPanel**) that will display inventory photos. We will use a graphic method to place a scaled version of any photo we load into the panel. As such, we need a special panel class with a **paintComponent** method (where the graphics methods are used). We define the **PhotoPanel** class using this code (added after the **HomeInventory** class):

```
class PhotoPanel extends JPanel
{
    public void paintComponent(Graphics g)
    {
        Graphics2D g2D = (Graphics2D) g;
        super.paintComponent(g2D);

        g2D.dispose();
    }
}
```

All graphics methods will go in the **paintComponent** method for this class. Add this class to your project.

Now, the panel properties are:

photoPanel:	
size	240, 160
gridx	4
gridy	7
gridwidth	3
insets	10, 0, 10, 10
anchor	CENTER

The panel is declared using:

```
PhotoPanel photoPanel = new PhotoPanel();
```

and added to the frame using:

```
photoPanel.setPreferredSize(new Dimension(240, 160));  
gridConstraints = new GridBagConstraints();
```

```
gridConstraints.gridx = 4;  
gridConstraints.gridy = 7;  
gridConstraints.gridwidth = 3;  
gridConstraints.insets = new Insets(10, 0, 10, 10);  
gridConstraints.anchor = GridBagConstraints.CENTER;  
getContentPane().add(photoPanel, gridConstraints);
```

Let's add a border to the panel. Add the shaded code to the panel **paintComponent** method.

```
class PhotoPanel extends JPanel  
{  
    public void paintComponent(Graphics g)  
    {  
        Graphics2D g2D = (Graphics2D) g;  
        super.paintComponent(g2D);  
  
        // draw border  
        g2D.setPaint(Color.BLACK);  
        g2D.draw(new Rectangle2D.Double(0, 0, getWidth() - 1, getHeight() - 1));  
  
        g2D.dispose();  
    }  
}
```

You need to add this import statement to the project (for the graphics code):

```
import java.awt.geom.*;
```

Run the project one more time to see the last control, the framed photo panel:

Home Inventory Manager

New

Delete

Save

Previous

Next

Print

Exit

Inventory Item

Location

Serial Number

Purchase Price

Date Purchased

Store/Website

Note

Photo

Marked?

Item Search

A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R
S	T	U	V	W	X
Y	Z				

At long last, all the controls are in place. We now start writing the code. There are many steps. First, let's address proper ordering of the controls for input.

Frame Design – Tab Order and Focus

This project has many controls for user input. We want to make sure it's clear to the user just what information is needed and when. We want the input to 'flow' from the top of the form to the bottom. Run the project. You will see that the **New** button has focus. We would prefer the cursor starting in the **itemTextField** control so the user can start typing input. And, if you tab through the controls, you will see that the toolbar buttons and search buttons (and other controls) can receive focus, even though we don't want this behavior.

Remove the ability to focus on all the toolbar buttons (**newButton**, **deleteButton**, **saveButton**, **previousButton**, **nextButton**, **printButton**, **exitButton**), the search buttons (**searchButton** array), the **markedCheckBox** and the **photoTextArea** controls. To do this, set the **focusable** property of each (use the **setFocusable** method) to **false**. This involves a single line of code for each control in the frame constructor code.

Now, run the project again. The tab sequence starts at the Inventory Item text field (**itemTextField**) and sequentially works down through all the input controls (text fields, combo box, date chooser) to the note text field (**noteTextField**), then the photo button (**photoButton**). We choose to skip the check box.

Another feature for the input is that whenever the user presses <Enter> after entering a value (in the combo box or text fields) or clicks a date in the date chooser, the control focus should move to the next control. This feature is implemented in the **ActionPerformed** methods for the combo box control and the five text fields and the **PropertyChange** method for the date control. Let's start at the top of the frame and work our way down, in proper tab order. First, the **itemTextFieldActionPerformed** method. Add a listener in the frame constructor code:

```
itemTextField.addActionListener(new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {
        itemTextFieldActionPerformed(e);
    }
});
```

Then add the corresponding method:

```
private void itemTextFieldActionPerformed(ActionEvent e)
{
    locationComboBox.requestFocus();
}
```

}
This method moves focus to the combo box (**locationComboBox**). Add it's listener:

```
locationComboBox.addActionListener(new ActionListener ()  
{  
    public void actionPerformed(ActionEvent e)  
    {  
        locationComboBoxActionPerformed(e);  
    }  
});
```

and corresponding method:

```
private void locationComboBoxActionPerformed(ActionEvent e)  
{  
    serialTextField.requestFocus();  
}
```

Focus moves to **serialTextField**. Add a listener:

```
serialTextField.addActionListener(new ActionListener ()  
{  
    public void actionPerformed(ActionEvent e)  
    {  
        serialTextFieldActionPerformed(e);  
    }  
});
```

Then, the corresponding method:

```
private void serialTextFieldActionPerformed(ActionEvent e)  
{  
    priceTextField.requestFocus();  
}
```

Focus moves to **priceTextField**. Add a listener:

```
priceTextField.addActionListener(new ActionListener ()
```

```

{
    public void actionPerformed(ActionEvent e)
    {
        priceTextFieldActionPerformed(e);
    }
});

```

Then, the corresponding method:

```

private void priceTextFieldActionPerformed(ActionEvent e)
{
    dateDateChooser.requestFocus();
}

```

Focus moves to **dateDateChooser**. Add a listener for property change:

```

dateDateChooser.addPropertyChangeListener(new PropertyChangeListener()
{
    public void propertyChange(PropertyChangeEvent e)
    {
        dateDateChooserPropertyChange(e);
    }
});

```

Then, the corresponding method:

```

private void
dateDateChooserPropertyChange(PropertyChangeEvent e)
{
    storeTextField.requestFocus();
}

```

Focus moves to **storeTextField**. Add a listener:

```

storeTextField.addActionListener(new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {

```

```
        storeTextFieldActionPerformed(e);  
    }  
});
```

Then, the corresponding method:

```
private void storeTextFieldActionPerformed(ActionEvent e)  
{  
    noteTextField.requestFocus();  
}
```

Focus moves to **noteTextField**. Add a listener:

```
noteTextField.addActionListener(new ActionListener ()  
{  
    public void actionPerformed(ActionEvent e)  
    {  
        noteTextFieldActionPerformed(e);  
    }  
});
```

And method:

```
private void noteTextFieldActionPerformed(ActionEvent e)  
{  
    photoButton.requestFocus();  
}
```

So, the focus ends up, as desired, on the photo button (**photoButton**). Double-check that all this added code is in the proper place. The listeners are created in the frame constructor code and the methods are added with all the other **HomeInventory** class methods.

Save and run the project. Press <**Enter**> in each control to make sure the focus transfers properly. One exception is the date chooser – you need to select a date to have the focus move to the next control. You will see this bit of work will be well worth it when you start entering information onto the form. We'll start writing code to process entries soon. First, let's look at how we'll structure all the information used to describe an inventory item.

Introduction to Object-Oriented Programming

Each inventory item requires nine individual pieces of information. Each item and the data type represented are:

- Description (**String** type)
- Location (**String** type)
- Marked indicator (**boolean** type)
- Serial number (**String** type)
- Purchase Price (**String** type)
- Purchase Date (**String** type)
- Purchase Location (**String** type)
- Note (**String** type)
- Photo file (**String** type)

One way to store all this information is to use nine different arrays, one for each quantity, each element of the array representing a single item in the inventory. Using arrays would be “doable,” but messy. It would be especially messy to write code for swapping inventory items (needed to make sure items remain in alphabetical order). Each swap would require swapping nine different array elements. And, what if we later want to add more information to an inventory item? We would need to remember everywhere these arrays were referenced in code to make the needed changes. There must be a better way to structure all this information. And there is.

We say Java is an **object-oriented** language. At this point in this *project*, we have used many of the built-in objects included with Java. We have used button objects, text field objects, label objects and many other controls. We have used graphics objects, stroke objects, paint objects, and shape objects. Having used these objects, we are familiar with such concepts as **declaring** an object, **constructing** an object and using an object’s **properties** and **methods**.

We have seen that objects are just things that have attributes (properties) with possible actions (methods). We’ll use the idea here to create our own “inventory item” objects. Our objects will only have properties (specified above) and no methods. You will see how creating such objects saves us lots of work.

Before getting started, you may be asking the question “If Java is an object-oriented language, why have we waited so long to start talking about using our own objects?” And, that’s a good question. Many books on Java dive right into building objects. We feel it’s best to see objects and use objects before trying to create your own. Java is a great language for doing this. The wealth of existing, built-in objects helps you learn about OOP before needing to build your own.

Now, let’s review some of the vocabulary of object-oriented programming. These are terms you’ve

seen before in working with the built-in objects of Java. A **class** provides a general description of an **object**. All objects are created from this class description. The first step in creating an object is adding a class to a Java project. **Note** the top line of every application has the keyword **class**.

The **class** provides a framework for describing three primary components:

- **Properties** – attributes describing the objects
- **Constructors** – methods that initialize the object
- **Methods** – procedures describing things an object can do

Once a class is defined, an object can be created or **instantiated** from the class. This simply means we use the class description to create a copy of the object we can work with. Once the instance is created, we **construct** the finished object for our use.

One last important term to define, related to OOP, is **inheritance**. This is a capability that allows one object to ‘borrow’ properties and methods from another object. This prevents the classic ‘reinventing the wheel’ situation. Inheritance is one of the most powerful features of OOP. In this project, we will only look at using a simple object, with just some properties.

Code Design – InventoryItem Class

The first step in creating our own object is to define the class from which the object will be created. This step (and all following steps) is best illustrated by example. And our example will be our inventory item object. We will be creating **InventoryItem** objects that have nine properties, one for each previously-listed piece of information input on the form.

We need to add a class to our project to allow the definition of our **InventoryItem** objects. We could add the class in the existing frame file. However, doing so would defeat a primary advantage of objects, that being re-use. Hence, we will create a separate file to hold our class. To do this, in **NetBeans**, right-click the project name (**Home Inventory**) and add a new Java class file to the project. Name that file **InventoryItem**. Put it in the **homeinventory** source folder. Delete the default code in the file and type these lines:

```
package homeinventory;  
  
public class InventoryItem  
{  
  
}
```

All code needed to define properties, constructors and methods for this class will be between the curly braces defining this class.

Add nine property declarations so the file looks like this:

```
package homeinventory;  
  
public class InventoryItem  
{  
  
    public String description;  
    public String location;  
    public boolean marked;  
    public String serialNumber;  
    public String purchasePrice;  
    public String purchaseDate;  
    public String purchaseLocation;  
    public String note;  
    public String photoFile;  
}
```

You should see how each property relates to the information on the frame. The keyword **public** indicates the variable is available to any class.

To declare an **InventoryItem** object named **myItem** (in our inventory application) use this line of code:

```
InventoryItem myItem;
```

To construct this object, use this line of code:

```
myItem = new InventoryItem();
```

This line just says “give me a new inventory item.” Our **InventoryItem** object is now complete, ready for use. This uses the **default** constructor automatically included with every class. The default constructor simply creates an object with no defined properties.

Once we have created an object, we can refer to properties using the usual notation:

```
objectName.propertyName
```

So, to set the **description** property of our example inventory item, we would use:

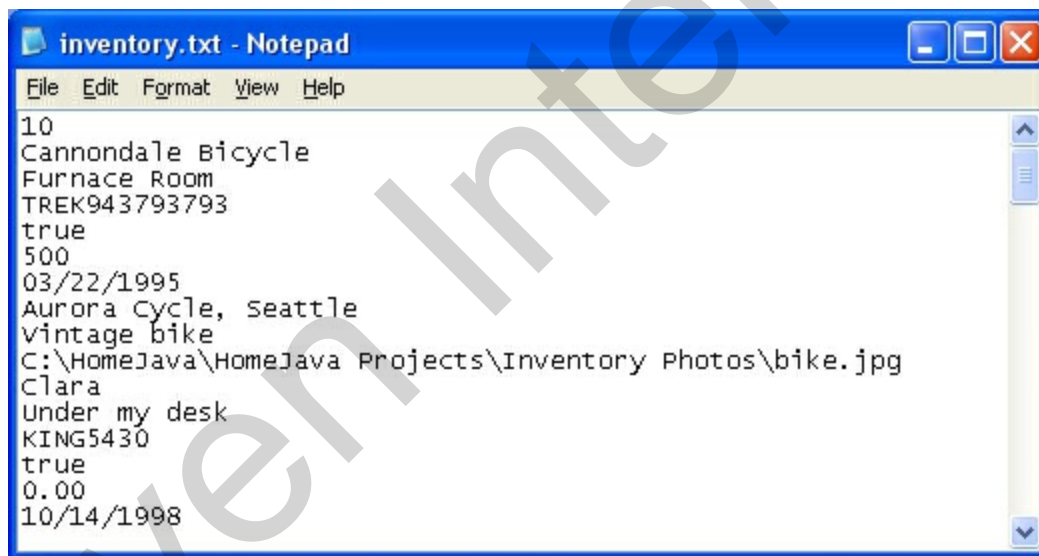
```
myItem.description = "This is my inventory item";
```

Let’s see how to use our **InventoryItem** class in the home inventory manager project. In this project, we will use an array of inventory items to keep track of things.

Code Design – Inventory File Input

We can now use the **InventoryItem** class to define how we will save inventory information in our project. First, we look at how to input that information from a file. We have chosen to store the inventory information in a built-in, “hard-wired” file. The program looks for a file named **inventory.txt** in the project folder. If the file can’t be found, the program begins with an empty inventory and, once items are added, these items are written to a new copy of **inventory.txt**.

The file (**inventory.txt**) keeps track of each item in the inventory. The file also stores the items in the combo box used to specify item location. In later code, we will look at how to modify these items, so new ones are saved. The **inventory.txt** file is sequential and stores one piece of information on each line. The first line is the number of inventory items in the file. After this line are 9 lines for each item in the inventory (each line saving one property for the **InventoryItem** object). After each item is described in the file is a line containing the number of items in the combo box control (**locationComboBox**). Following this line are the corresponding combo box items. In the **\HomeJava\HomeJava Projects** folder is the sample provided with these notes (seen in the project preview). Open this file in a text editor and you will see:

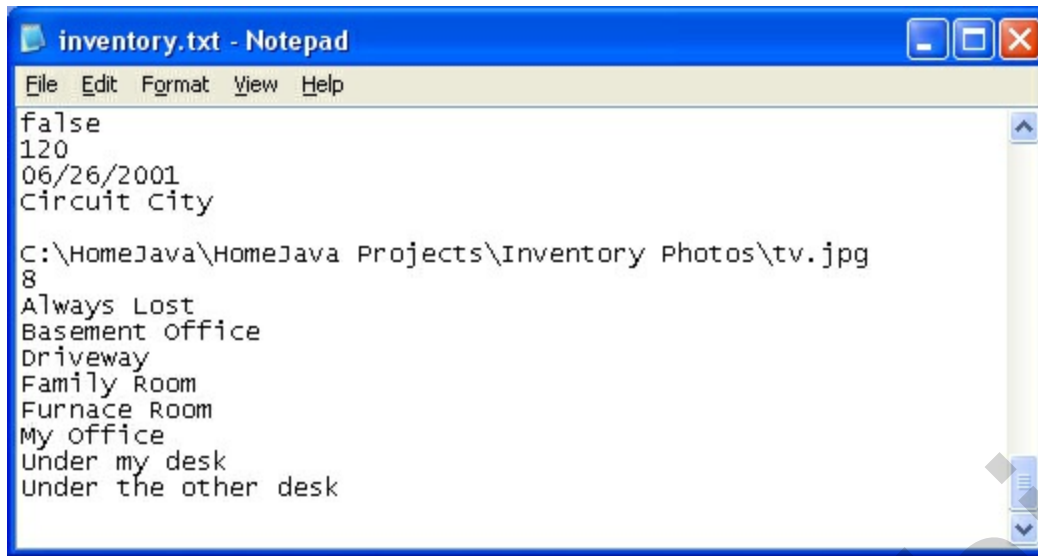


```
File Edit Format View Help
10
Cannondale Bicycle
Furnace Room
TREK943793793
true
500
03/22/1995
Aurora Cycle, Seattle
Vintage bike
C:\HomeJava\HomeJava Projects\Inventory Photos\bike.jpg
Clara
Under my desk
KING5430
true
0.00
10/14/1998
```

You see this file has 10 entries and there will be 9 lines per entry. You can see the first entry (**Cannondale Bicycle**) and the beginning of the second (**Clara**).

Scroll down to the bottom to see:

Note : Create your own **inventory.txt** file. This is just a sample file for demonstrating the project.



```
inventory.txt - Notepad
File Edit Format View Help
false
120
06/26/2001
Circuit City

C:\HomeJava\HomeJava Projects\Inventory Photos\tv.jpg
8
Always Lost
Basement Office
Driveway
Family Room
Furnace Room
My office
Under my desk
Under the other desk
```

After the last entry are the eight (8) items used in the combo box control. Let's write the code to read this file.

When the home inventory manager project begins, the program should read in the **inventory.txt** file to store the inventory item information. Here's where we'll use the **InventoryItem** object. The steps are:

- Open **inventory.txt** for input.
- Read in the number of entries.
- For each entry in the file:
 - o Create a new **InventoryItem** object.
 - o Read in the nine object properties.
- When done reading inventory items, read in the number of items in combo box control.
- Read in combo box items.
- Close file.

Make sure you are now working with the **HomeInventory.java** file. We will use the file objects to read the file, so add this import statement at the top of the code window:

```
import java.io.*;
```

The code associated with the above steps goes at the end of the frame constructor method so it is executed when the program first begins. Define three class level variables:

```
final int maximumEntries = 300;
```

```
int numberEntries;
```

```
InventoryItem[] myInventory = new InventoryItem[maximumEntries];
```

maximumEntries is the maximum number of inventory items allowed, **numberEntries** is the

number of entries in our inventory and **myInventory** is a 0-based array of **InventoryItem** objects used to store the information. With these variables, the code to open and read the **inventory.txt** file is:

```
int n;
// open file for entries try
{
    BufferedReader inputFile = new BufferedReader(new FileReader("inventory.txt"));
    numberEntries =
Integer.valueOf(inputFile.readLine()).intValue();
    if (numberEntries != 0)
    {
        for (int i = 0; i < numberEntries; i++)
        {
            myInventory[i] = new InventoryItem();
            myInventory[i].description = inputFile.readLine();
            myInventory[i].location = inputFile.readLine();
            myInventory[i].serialNumber = inputFile.readLine();
            myInventory[i].marked =
Boolean.valueOf(inputFile.readLine()).booleanValue();
            myInventory[i].purchasePrice = inputFile.readLine();
            myInventory[i].purchaseDate = inputFile.readLine();
            myInventory[i].purchaseLocation = inputFile.readLine();
            myInventory[i].note = inputFile.readLine();
            myInventory[i].photoFile = inputFile.readLine();
        }
    }
    // read in combo box elements
    n = Integer.valueOf(inputFile.readLine()).intValue();
    if (n != 0)
    {
        for (int i = 0; i < n; i++)
        {
            locationComboBox.addItem(inputFile.readLine());
        }
    }
}
```

```
        inputFile.close();
    }
    catch (Exception ex)
    {
        numberEntries = 0;
    }
    if (numberEntries == 0)
    {
        newButton.setEnabled(false);
        deleteButton.setEnabled(false);
        nextButton.setEnabled(false);
        previousButton.setEnabled(false);
        printButton.setEnabled(false);
    }
}
```

Let's take a look at this code. All the code is in a **try/catch** structure in case the input file cannot be opened. If the file is successfully opened, we read **numberEntries**, then read each subsequent entry, creating a new **InventoryItem** object for each entry. Once the inventory items have been input, the combo box items are read in. If **numberEntries** is zero when done (meaning the file couldn't be opened or truly had zero elements), we set the toolbar buttons so only a new item can be entered. Add this method to your project.

Let's try this code. Copy your **inventory.txt** file into your project folder. Save and run the project. If the file opens and reads successfully, you should see a blank form with all toolbar buttons enabled (try clicking the drop-down in the combo box to see the items added). Here's my frame:

Home Inventory Manager

Inventory Item

Location ☐ **Marked?**

Serial Number

Purchase Price

Store/Website

Note

Photo

Item Search

A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R
S	T	U	V	W	X
Y	Z				

If only the **Save** button is enabled, there was an error in reading the file. If this occurs, make sure the file is in the correct folder and double-check the code.

Please Note : The location shown in the combo-box entries is only for demonstration purpose. You may add entries appropriate your home / office.

Code Design – Viewing Inventory Item

We can now read in the input file, but it's not very satisfying not being able to see the results. We remedy that now by writing code to display the properties of an inventory item (including the photo). The code simply sets the correct frame control with the corresponding property.

We will use a general method **showEntry** to display the properties of a single inventory item. The method will have an **int** argument, specifying the entry number (from **1** to **numberEntries**) to display. The method is:

```
private void showEntry(int j)
{
    // display entry j (1 to numberEntries)
    itemTextField.setText(myInventory[j - 1].description);
    locationComboBox.setSelectedItem(myInventory[j - 1].location);
    markedCheckBox.setSelected(myInventory[j - 1].marked);
    serialTextField.setText(myInventory[j - 1].serialNumber);
    priceTextField.setText(myInventory[j - 1].purchasePrice);
    dateDateChooser.setDate(stringToDate(myInventory[j - 1].purchaseDate));
    storeTextField.setText(myInventory[j - 1].purchaseLocation);
    noteTextField.setText(myInventory[j - 1].note);
    showPhoto(myInventory[j - 1].photoFile);
    itemTextField.requestFocus();
}
```

This code simply transfers the properties of the **myInventory InventoryItem** object into the appropriate controls. When done, it gives focus to the first text field (**itemTextField**).

The **showEntry** method uses a general method, **stringToDate**. We store the purchase date as a **String** type in the data file. We use a **month/day/year** representation, always using two digits for the month and day. The date chooser control requires dates to be of **Date** type. Hence, we need the capability of converting strings to dates, and dates to strings. The **stringToDate** and **dateToString** methods that do this task are:

```
private Date stringToDate(String s)
{
    int m = Integer.valueOf(s.substring(0, 2)).intValue() - 1;
    int d = Integer.valueOf(s.substring(3, 5)).intValue();
    int y = Integer.valueOf(s.substring(6)).intValue() - 1900;
```

```

        return(new Date(y, m, d));
    }

    private String dateToString(Date dd)
    {
        String yString = String.valueOf(dd.getYear() + 1900);
        int m = dd.getMonth() + 1;
        String mString = new DecimalFormat("00").format(m);
        int d = dd.getDate();
        String dString = new DecimalFormat("00").format(d);
        return(mString + "/" + dString + "/" + yString);
    }

```

These routines require these import statements:

```

import java.util.*;
import java.text.*;

```

To show a photo, we use another general method **showPhoto**. The code to actually display the photo goes in the **paintComponent** method for the **PhotoPanel** class. We will write code for this method soon. The **showPhoto** method simply accepts the photo file name (**photoFile**) as a **String** argument and displays the name. A **try/catch** structure is used in case there is an error loading a photo file:

```

private void showPhoto(String photoFile)
{
    if (!photoFile.equals(""))
    {
        try
        {
            photoTextArea.setText(photoFile);
        }
        catch (Exception ex)
        {
            photoTextArea.setText("");
        }
    }
    else
    {

```

```

        photoTextArea.setText("");
    }
    photoPanel.repaint();
}

```

Place all of these methods (**showEntry**, **stringToDate**, **dateToString** and **showPhoto**) in your project. Add the two new **import** statements.

We need to modify the frame constructor code to call the display routine. First, add another class level variable declaration:

```

int currentEntry;

```

currentEntry will always point to the current entry in the inventory file (going from **1** to **numberEntries**). The modified code that uses this variable to display the entry is (changes are shaded, much unmodified code is not shown):

```

int n;
// open file for entries
try
{
    BufferedReader inputFile = new BufferedReader(new FileReader("inventory.txt"));
    .
    .
}
inputFile.close();
    currentEntry = 1;
    showEntry(currentEntry);
}
catch (Exception ex)
{
    numberEntries = 0;
    currentEntry = 0;
}
if (numberEntries == 0)
{
    newButton.setEnabled(false);
    deleteButton.setEnabled(false);
}

```



```

nextButton.setEnabled(false);
previousButton.setEnabled(false);
printButton.setEnabled(false);
}

```

Make the noted changes.

Save and run the project. You should now see the first item in the inventory displayed (except for photo):

The screenshot shows a Java Swing window titled "Home Inventory Manager". On the left is a blue sidebar with buttons: New, Delete, Save, Previous, Next, Print, and Exit. The main area contains a form for an inventory item. The fields are filled with the following data:

- Inventory Item: Cannondale Bicycle
- Location: Furnace Room (dropdown menu)
- Serial Number: TREK943793793
- Purchase Price: 500
- Date Purchased: Mar 22, 1995
- Store/Website: Aurora Cycle, Seattle
- Note: Vintage bike
- Photo: C:\HomeJava\HomeJava Projects\Inventory Photos\bike.jpg

Below the form is an "Item Search" section with a grid of letters A through Z. The letters are arranged in a 5x6 grid, with the last row containing only Y and Z.

A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R
S	T	U	V	W	X
Y	Z				

Code Design – Viewing Photo

The photo is displayed in **photoPanel** (using code in the **paintComponent** method). We use the **drawImage** graphics method to display the photo. This method allows us to scale any photo to fit within the display region. If our graphics region is **g2D**, the **drawImage** method is:

```
g2D.drawImage(myImage, x, y, w, h, null);
```

In this method, the image (**myImage**) will be positioned at (**x, y**) with width **w** and height **h**. We will adjust the width and height arguments to maintain the width-to-height ratio of the corresponding photo.

A word about photo location. The example file assumes the inventory photos are located in a folder named **c:\HomeJava\HomeJava Projects\Inventory Photos**. Your copy of the photos may or may not be in such a folder. If they are in such a folder, great! If not, you have a few choices: (1) create such a folder and copy the photos to that folder, (2) ignore the location and let the program run, knowing the photos won't display. **You choose**. Once the program is fully functional, you can load each picture individually from folders on your computer. Then, when the data file is saved, those locations are saved correctly for your machine.

Add the shaded code to the **PhotoPanel** class **paintComponent** method:

```
public void paintComponent(Graphics g)  
{  
    Graphics2D g2D = (Graphics2D) g;  
    super.paintComponent(g2D);  
  
    // draw border  
    g2D.setPaint(Color.BLACK);  
    g2D.draw(new Rectangle2D.Double(0, 0, getWidth() - 1, getHeight() - 1));  
  
    // show photo  
    Image photoImage = new  
ImageIcon(HomeInventory.photoTextArea.getText()).getImage( );  
    int w = getWidth();  
    int h = getHeight();  
    double rWidth = (double) getWidth() / (double) photoImage.getWidth(null);  
    double rHeight = (double) getHeight() / (double) photoImage.getHeight(null);  
    if (rWidth > rHeight)
```

```

{
    // leave height at display height, change width by amount height is changed
    w = (int) (photoImage.getWidth(null) * rHeight);
}
else
{
    // leave width at display width, change height by amount width is changed
    h = (int) (photoImage.getHeight(null) * rWidth);
}
// center in panel
g2D.drawImage(photoImage, (int) (0.5 * (getWidth() - w)), (int) (0.5 * (getHeight() - h)),
w, h, null);

g2D.dispose();
}

```

In this code, we first create an **Image** object (**photoImage**) from the photo file. We then determine photo scaling so that the width-to-height ration of the original photo is maintained. Lastly, we use the **drawImage** method to position the photo on the **photoPanel**. Note the code requires access to the **photoTextArea** control in the **HomeInventory** class. Because of this, the **photoTextArea** declaration must be prefaced with **static**:

```
static JTextArea photoTextArea = new JTextArea();
```

Make this simple change.

Run the project. You should see the first inventory item (including photo, assuming you solved any “photo location” problem you may have had):

Home Inventory Manager

Inventory Item Cannondale Bicycle

Location Furnace Room ☒ **Marked?**

Serial Number TREK943793793

Purchase Price 500 **Date Purchased** Mar 22, 1995


Store/Website Aurora Cycle, Seattle

Note Vintage bike

Photo C:\HomeJava\HomeJava Projects\Inventory Photos\bike.jpg

Item Search

A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R
S	T	U	V	W	X
Y	Z				



At this point, we would like to be able to move to the next item, or move backward. Let's write the code to do that.

Code Design – Item Navigation

First, we modify the **showEntry** method to establish proper **enabled** properties for the two toolbar buttons (**previousButton** and **nextButton**) used to move among the inventory items. We set these properties based on whether we're at the beginning, at the end or in the middle of the item list (changes are shaded):

```
private void showEntry(int j)
{
    // display entry j (1 to numberEntries)
    itemTextField.setText(myInventory[j - 1].description);
    locationComboBox.setSelectedItem(myInventory[j - 1].location);
    markedCheckBox.setSelected(myInventory[j - 1].marked);
    serialTextField.setText(myInventory[j - 1].serialNumber);
    priceTextField.setText(myInventory[j - 1].purchasePrice);
    dateDateChooser.setDate(stringToDate(myInventory[j - 1].purchaseDate));
    storeTextField.setText(myInventory[j - 1].purchaseLocation);
    noteTextField.setText(myInventory[j - 1].note);
    showPhoto(myInventory[j - 1].photoFile);
    nextButton.setEnabled(true);
    previousButton.setEnabled(true);
    if (j == 1)
        previousButton.setEnabled(false);
    if (j == numberEntries)
        nextButton.setEnabled(false);
    itemTextField.requestFocus();
}
```

Make these changes.

Now, we need code for the **ActionPerformed** methods on the **previousButton** and **nextButton** buttons. In each case, we adjust **currentEntry** in the proper direction and display the item. The methods are:

```
private void previousButtonActionPerformed(ActionEvent e)
{
    currentEntry--;
```

```

    showEntry(currentEntry);
}

```

```

private void nextButtonActionPerformed(ActionEvent e)
{
    currentEntry++;
    showEntry(currentEntry);
}

```

Add these methods to your project.

And, while we're at it, add the **exitButtonActionPerformed** method to your project:

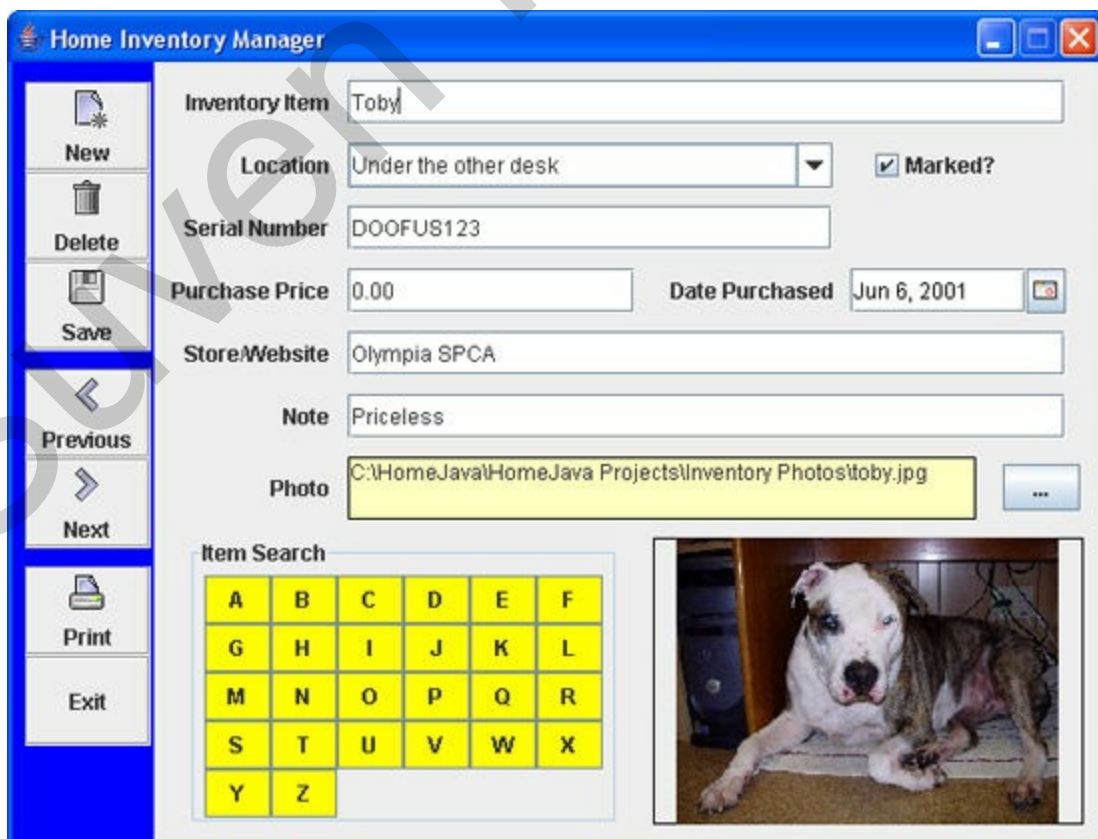
```

private void exitButtonActionPerformed(ActionEvent e)
{
    exitForm(null);
}

```

We call the **exitForm** method.

Again, save and run the project. You should now be able to view all 10 items in the sample file by using the **Previous** and **Next** buttons in the toolbar. Give it a try. Here's my dog Toby:



Try the **Exit** button.

Suven Internship

Code Design – Inventory File Output

If inventory entries are edited or new items added (we'll see how to do this next), we want to save all entries back to the **inventory.txt** file. We do this output in the **exitForm** method. That method is essentially the same as the code in the frame constructor with the **readLine** lines replaced by **println** statements:

```
private void exitForm(WindowEvent evt)
{
    // write entries back to file
    try
    {
        PrintWriter outputFile = new PrintWriter(new BufferedWriter(new
FileWriter("inventory.txt")));
        outputFile.println(numberEntries);
        if (numberEntries != 0)
        {
            for (int i = 0; i < numberEntries; i++)
            {
                outputFile.println(myInventory[i].description);
                outputFile.println(myInventory[i].location);
                outputFile.println(myInventory[i].serialNumber);
                outputFile.println(myInventory[i].marked);
                outputFile.println(myInventory[i].purchasePrice);
                outputFile.println(myInventory[i].purchaseDate);

                outputFile.println(myInventory[i].purchaseLocation);
                outputFile.println(myInventory[i].note);
                outputFile.println(myInventory[i].photoFile);
            }
        }
        // write combo box entries
        outputFile.println(locationComboBox.getItemCount());
        if (locationComboBox.getItemCount() != 0)
        {
            for (int i = 0; i < locationComboBox.getItemCount(); i++)
                outputFile.println(locationComboBox.getItemAt(i));
        }
    }
}
```



```
    }  
    outputFile.close();  
}  
catch (Exception ex)  
{  
  
}  
System.exit(0);  
}
```

All but the last line is new code. Add this method to your project. Note the code to write the combo box items back to file also.

Save and run the project. The input file will be read and the items displayed. Stop the project. The file will be written back to disk. Currently, the same file will be written back. This is because we have no editing capability.

Code Design – Input Validation

Before adding editing capability, we need to address a couple of input validation issues. First, let's discuss the **purchasePrice** property. We could make sure this is a numeric input, using validation code used in the loan assistant project. I choose not to do this for a couple of reasons. First, the value is never used with any math functions, so we really don't care if it's a number. Second, it allows the user to write something like 'Free' or 'Gift' in the text field and have it properly saved.

Next, we address the **location** property. The value for this property is obtained from the **locationComboBox** control. Two selection possibilities exist: (1) choose from an existing location, (2) type in a new location. If a new location is typed in, it is added to the list portion of the combo box, so it can be saved for future edits (in the **inventory.txt** file). So, we need code to check a typed entry versus the existing list to see if the new entry needs to be added to the list box. The code to do this goes in the **locationComboBoxActionPerformed** method. Make the shaded changes to this method:

```
private void locationComboBoxActionPerformed(ActionEvent e)
{
    // If in list - exit method
    if (locationComboBox.getItemCount() != 0)
    {
        for (int i = 0; i < locationComboBox.getItemCount(); i++)
        {
            if
(locationComboBox.getSelectedItem().toString().equals(locationComboBox.getItemAt(i).toString()))
            {
                serialTextField.requestFocus();
                return;
            }
        }
    }
    // If not found, add to list box

    locationComboBox.addItem(locationComboBox.getSelectedItem( ));
    serialTextField.requestFocus();
}
```

You should be able to follow the logic of what's going on here. Add this method to the project.

Lastly, we address how to load a photo into the photo panel control on the frame. When the user clicks the button with an ellipsis (**photoButton**) next to the **Photo** text area control, the open file dialog control (using **JChooser**) should appear allowing the user to select a file. The **FileNameExtensionFilter** method is used to set the filter. Add this import statement:

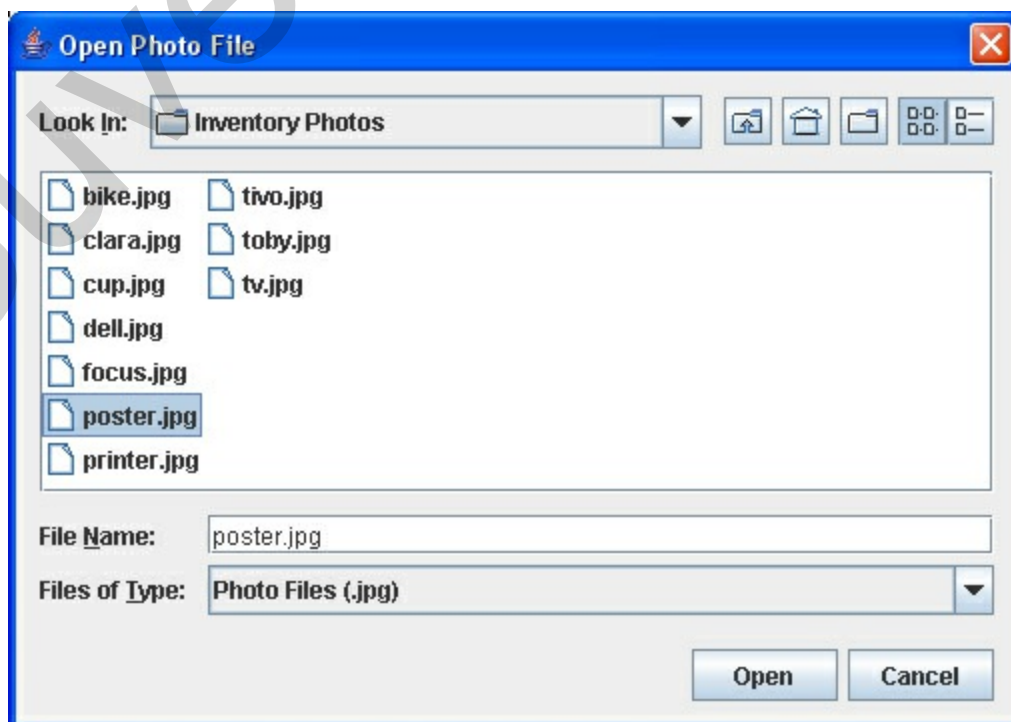
```
import javax.swing.filechooser.*;
```

Once a file is selected, the photo panel control should display the photo. We have already written code to do most of these steps in the **showPhoto** method we use to display an already stored photo file. So, the **photoButtonActionPerformed** method is:

```
private void photoButtonActionPerformed(ActionEvent e)
{
    JFileChooser openChooser = new JFileChooser();
    openChooser.setDialogType(JFileChooser.OPEN_DIALOG);
    openChooser.setDialogTitle("Open Photo File");
    openChooser.addChoosableFileFilter(new FileNameExtensionFilter("Photo Files",
"jpg"));
    if (openChooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION)
        showPhoto(openChooser.getSelectedFile().toString());
}
```

Add this method to the project.

Save and run the project to make sure the code compiles. You can try the new validations with the existing inventory items. Click the photo button. The dialog will appear:



You can change a photo, if you choose. Any changes to an item won't be saved (changes to the combo box will be saved). We now add editing capability to allow saving changes to existing and new inventory items.

Suven Internship

Code Design – New Inventory Item

When the project first begins (with an empty input file) or when the user clicks the **New** button in the toolbar, we want the form to be in a state to accept a new set of inventory information. The steps are:

- Disable all toolbar buttons, except **saveButton**.
- Blank out all text field controls and combo box.
- Uncheck check box control (**markedCheckBox**).
- Set calendar to today's date.
- Blank out **photoPanel** panel and **photoTextField** label.
- Give **itemTextField** focus.

The code for these steps is placed in a general method **blankValues**. We use a method because it is needed here, to start a new item, and later in the delete method (in case we delete the last item in the inventory). The method to implement the above steps is:

```
private void blankValues()
{
    // blank input screen
    newButton.setEnabled(false);
    deleteButton.setEnabled(false);
    saveButton.setEnabled(true);
    previousButton.setEnabled(false);
    nextButton.setEnabled(false);
    printButton.setEnabled(false);
    itemTextField.setText("");
    locationComboBox.setSelectedItem("");
    markedCheckBox.setSelected(false);
    serialTextField.setText("");
    priceTextField.setText("");
    dateDateChooser.setDate(new Date());
    storeTextField.setText("");
    noteTextField.setText("");
    photoTextArea.setText("");
    photoPanel.repaint();
    itemTextField.requestFocus();
}
```

Add this method to your project.

With this general method, the **newButton Click** event is coded simply as:

```
private void newButtonActionPerformed(ActionEvent e)
{
    blankValues();
}
```

Add this method to your project.

Now we need code to save entries for a new inventory item. When the user clicks the **Save** button on the toolbar, the following steps occur:

- Make sure there is an entry in **itemTextField** (the only required input). Capitalize the first character (to insure proper ordering).
- Increment **numberEntries**.
- Determine entry location in **myInventory** array (alphabetically, using **itemTextField Text** property)
- Once location is determined, move all items “below” location down one position in **myInventory** array.
- Establish properties for new array entry.
- Display new entry.
- Disable **newButton**, if we’ve reached the maximum number of entries.
- Enable **deleteButton** and **printButton**.

The tricky part of the code associated with these steps involves moving elements in the **myInventory** array. Let me explain. With normal Java variables **a**, **b**, **c** if you write:

```
a = b;
b = c;
```

a will be replaced by the value in **b**. When **b** is replaced by **c**, **a** is unchanged, retaining the original value for **b**.

What if **a**, **b**, **c** are objects (such as elements of the **myInventory** array) and we write the same code:

```
a = b;
b = c;
```

With objects, **a** will be assigned the same memory location as **b**, not a copy of its value. When **b** is

then assigned to c, a will also change to c since it shares the same memory location. To avoid this, we need one additional step following the assignment of b to a. The modified code is:

```
a = b;  
b = new Object();  
b = c;
```

In this code, once b is assigned to a, we create a new object for b, giving it a new memory location prior to assigning it to c. This “breaks” the connection between memory locations for a and b. This modified code gives us the desired result of a having the original value for b and b having the new value for c.

The code for saving an entry is placed in the **saveButtonActionPerformed** method:

```
private void saveButtonActionPerformed(ActionEvent e)  
{  
    // check for description  
    itemTextField.setText(itemTextField.getText().trim());  
    if (itemTextField.getText().equals(""))  
    {  
        JOptionPane.showConfirmDialog(null, "Must have item description.", "Error",  
JOptionPane.DEFAULT_OPTION, JOptionPane.ERROR_MESSAGE);  
        itemTextField.requestFocus();  
        return;  
    }  
    // capitalize first letter  
    String s = itemTextField.getText();  
    itemTextField.setText(s.substring(0, 1).toUpperCase() + s.substring(1));  
    numberEntries++;  
    // determine new current entry location based on description  
    currentEntry = 1;  
    if (numberEntries != 1)  
    {  
        do  
        {  
            if  
(itemTextField.getText().compareTo(myInventory[currentEntry - 1].description) < 0)  
                break;  
        }  
    }
```

```

        currentEntry++;
    }
    while (currentEntry < numberEntries);
}
// move all entries below new value down one position unless at end
if (currentEntry != numberEntries)
{
    for (int i = numberEntries; i >= currentEntry + 1; i--)
    {
        myInventory[i - 1] = myInventory[i - 2];
        myInventory[i - 2] = new InventoryItem();
    }
}
myInventory[currentEntry - 1] = new InventoryItem();
myInventory[currentEntry - 1].description = itemTextField.getText();
myInventory[currentEntry - 1].location =
locationComboBox.getSelectedItem().toString();
myInventory[currentEntry - 1].marked = markedCheckBox.isSelected();
myInventory[currentEntry - 1].serialNumber = serialTextField.getText();
myInventory[currentEntry - 1].purchasePrice = priceTextField.getText();
myInventory[currentEntry - 1].purchaseDate =
dateToString(dateDateChooser.getDate());
myInventory[currentEntry - 1].purchaseLocation = storeTextField.getText();
myInventory[currentEntry - 1].photoFile = photoTextArea.getText();
myInventory[currentEntry - 1].note = noteTextField.getText();
showEntry(currentEntry);
if (numberEntries < maximumEntries)
    newButton.setEnabled(true);
else
    newButton.setEnabled(false);
deleteButton.setEnabled(true);
printButton.setEnabled(true);
}

```

Study this code. If there is no entry in **itemTextField**, a message box is displayed. If there is an entry, capitalize the first character, to obtain proper ordering. We then determine the location of the new entry in the list of current entries. Once that position (**currentEntry**) is found, all other array elements

are moved down one position (paying attention to how we “equate” objects). A new **InventoryItem** is created at **currentEntry - 1** (recall we’re using a 0-based array) and the control values placed in the appropriate object properties. Based on the number of entries, toolbar button status is modified. Add this method to your project.

One option the user has while adding a new inventory item is to click on the **Exit** button in the toolbar, essentially stopping the program. We want to add a message box to the program to make sure if this happens, the user really means to exit. This message box could be placed in the **exitButtonActionPerformed** event (which closes the form, invoking the **exitForm** method). Recall, however, a user can also exit a program by clicking the **X** in the upper right corner of the form, also invoking the **exitForm** method. So, to intercept all exit requests, we place this new message box in the **exitForm** method. The modified method is (changes are shaded, unmodified code writing properties back to file is not shown):

```
private void exitForm(WindowEvent evt)
{
    if (JOptionPane.showConfirmDialog(null, "Any unsaved changes will be lost.\nAre you
sure you want to exit?", "Exit Program", JOptionPane.YES_NO_OPTION,
JOptionPane.QUESTION_MESSAGE) == JOptionPane.NO_OPTION)
        return;

    // write entries back to file
    try
    {
        PrintWriter outputFile = new PrintWriter(new BufferedWriter(new
FileWriter("inventory.txt")));
        outputFile.println(numberEntries);
        .
        .
    }
}
```

We need to make one change to make this code work. By default, when this method is called, the frame is closed and cannot be reopened. To avoid this, add this line of code in the frame constructor when the frame is first created:

```
setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

Add these code modifications to your project.

Save and run the project. You should now have the capability to add and save a new item to the inventory. Let’s try it. Click the **New** toolbar button to see a blank form ready for input (only the **Save** button and **Exit** buttons are enabled):

Home Inventory Manager

Inventory Item:

Location: ☐ Marked?

Serial Number:

Purchase Price: Date Purchased: Nov 30, 2008

Store/Website:

Note:

Photo:

Item Search:

A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R
S	T	U	V	W	X
Y	Z				

Type in some entries. Try the combo box selections (type a new one if you want). Add a photo if you have one. When done, click **Save** to make sure the item is properly sorted.

I added a **Couch** to my inventory (no photo). After clicking **Save**, I see:

Home Inventory Manager

Inventory Item: Couch

Location: Living Room ☐ Marked?

Serial Number: None

Purchase Price: Free Date Purchased: Nov 30, 2008

Store/Website: Goodwill

Note:

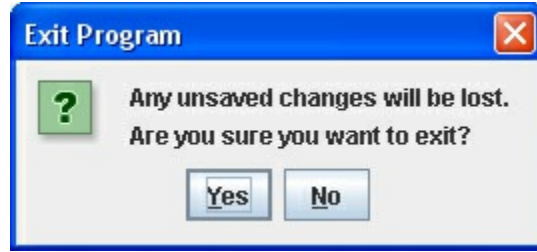
Photo:

Item Search:

A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R
S	T	U	V	W	X
Y	Z				

Notice all toolbar buttons are now active. I can add another item or move to another item. By

clicking **Previous** and **Next**, I can see that the item is properly located in the list (right after my **Coffee Cup**). Stop the project when you want. When you click **Exit**, you should see the message box we added to prevent inadvertent stopping of the program:



Make sure both the **Yes** and **No** options work correctly. Stop, then rerun the project. Make sure any added items are now in the inventory.

Code Design – Deleting Inventory Items

After entering a new item (or when viewing an existing item), the **Delete** toolbar button is enabled, but there is no code “behind” the button. Let’s write that code.

When a user clicks the **Delete** button while displaying an entry, the following should happen:

- Ask the user if he/she really wants to delete the entry.
- Move all items “below” displayed entry up one position in **myInventory** array. This removes the entry from the array.
- Decrement **numberEntries**.
- If entry deleted is last item, set form up for new entry.
- If more entries remain after deletion, display entry preceding deleted entry.

The code for the above steps is placed in the **deleteButtonActionPerformed** method:

```
private void deleteButtonActionPerformed(ActionEvent e)
{
    if (JOptionPane.showConfirmDialog(null, "Are you sure you want to delete this item?",
    "Delete Inventory Item", JOptionPane.YES_NO_OPTION,
    JOptionPane.QUESTION_MESSAGE) == JOptionPane.NO_OPTION)
        return;
    deleteEntry(currentEntry);
    if (numberEntries == 0)
    {
        currentEntry = 0;
        blankValues();
    }
    else
    {
        currentEntry--;
        if (currentEntry == 0)
            currentEntry = 1;
        showEntry(currentEntry);
    }
}
```

Notice if we delete the last item in the inventory, the form is ‘blanked.’ Otherwise, the entry preceding

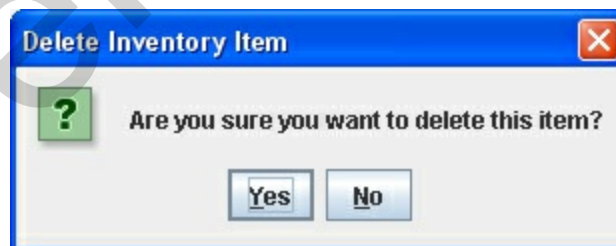
the deleted entry (if there is one) is displayed. Add this method to your project.

The above code uses a general method **deleteEntry** to remove an entry from the **myInventory** array. The code for this method is (the **int** argument indicates which item to remove):

```
private void deleteEntry(int j)
{
    // delete entry j
    if (j != numberEntries)
    {
        // move all entries under j up one level
        for (int i = j; i < numberEntries; i++)
        {
            myInventory[i - 1] = new InventoryItem();
            myInventory[i - 1] = myInventory[i];
        }
    }
    numberEntries--;
}
```

Again, notice the special way to “equate” objects. Add this method to your project.

Save and run the project with these changes. Make sure you can delete any entries you added earlier. You will see this message box when you try to delete an entry:



Make sure both the **Yes** and **No** options work. I was able to successfully delete my **Couch** from the inventory.

Code Design – Editing Inventory Items

There is one problem you may notice. If you edit a current entry in the inventory and click **Save**, a new item is added to the inventory, rather than a simple update of the existing item. We need to modify the save method to be able to handle editing an existing item.

The approach we take is that if we are editing an existing item and click **Save**, we first delete it, then treat the modified item as if it is a new item. This allows us to use the existing save code and also properly sorts the edited item (if the **Description** property changed). The modified **saveButtonActionPerformed** method is (changes are shaded, much unmodified code not shown):

```
private void saveButtonActionPerformed(ActionEvent e)
{
    // check for description
    itemTextField.setText(itemTextField.getText().trim());
    if (itemTextField.getText().equals(""))
    {
        JOptionPane.showConfirmDialog(null, "Must have item description.", "Error",
        JOptionPane.DEFAULT_OPTION, JOptionPane.ERROR_MESSAGE);
        itemTextField.requestFocus();
        return;
    }
    if (newButton.isEnabled())
    {
        // delete edit entry then resave
        deleteEntry(currentEntry);
    }
    .
    .
}
```

In this code, we use the **enabled** status of **newButton** to determine if we are saving an existing item (**enabled** is **true**) or a new item (**enabled** is **false**). Add these changes to your project.

Save and run the project. You now have full editing capability in the home inventory manager project. You can view inventory items, add new items to your inventory, delete items, or modify existing items. All information can now be properly saved.

We still need to add search and print capabilities to our project. But, first we need to address one

“small” annoyance. If you edit an existing item and then click **New**, **Previous**, or **Next** without clicking **Save**, your changes are lost. (You can also click **Exit**, but we have already added code for that event to give the user a chance to reconsider). It would be nice if the program would “save us from ourselves” and ask us if we’d like to save the changes before moving on. This is a straightforward modification. We essentially need to know if anything was changed for a particular item. If changes were made and we attempt to move away from that item (click **New**, **Previous** or **Next**) without clicking **Save**, we can display a message box asking if the changes should be saved.

We will use a general method, **checkSave**, to see if a save might be needed. Modify the **newButton**, **previousButton** and **nextButton** **ActionPerformed** methods to call this method (changes are shaded):

```
private void newButtonActionPerformed(ActionEvent e)
{
    checkSave();
    blankValues();
}

private void previousButtonActionPerformed(ActionEvent e)
{
    checkSave();
    currentEntry--;
    showEntry(currentEntry);
}

private void nextButtonActionPerformed(ActionEvent e)
{
    checkSave();
    currentEntry++;
    showEntry(currentEntry);
}
```

The general method **checkSave** used by each of these methods is:

```
private void checkSave()
{
    boolean edited = false;
    if (!myInventory[currentEntry - 1].description.equals(itemTextField.getText()))
        edited = true;
}
```

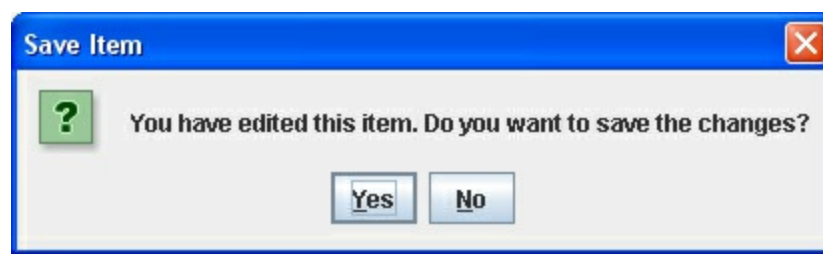
```

    else if (!myInventory[currentEntry -
1].location.equals(locationComboBox.getSelectedItem().toString()))
        edited = true;
    else if (myInventory[currentEntry - 1].marked != markedCheckBox.isSelected())
        edited = true;
    else if (!myInventory[currentEntry - 1].serialNumber.equals(serialTextField.getText()))
        edited = true;
    else if (!myInventory[currentEntry - 1].purchasePrice.equals(priceTextField.getText()))
        edited = true;
    else if (!myInventory[currentEntry -
1].purchaseDate.equals(dateToString(dateDateChooser.getDate())))
        edited = true;
    else if (!myInventory[currentEntry -
1].purchaseLocation.equals(storeTextField.getText()))
        edited = true;
    else if (!myInventory[currentEntry - 1].note.equals(noteTextField.getText()))
        edited = true;
    else if (!myInventory[currentEntry - 1].photoFile.equals(photoTextArea.getText()))
        edited = true;
    if (edited)
    {
        if (JOptionPane.showConfirmDialog(null, "You have edited this item. Do you want to
save the changes?", "Save Item", JOptionPane.YES_NO_OPTION,
JOptionPane.QUESTION_MESSAGE) == JOptionPane.YES_OPTION)
            saveButton.doClick();
    }
}

```

This method has a local **boolean** variable **edited** that tells us if the current inventory item has been modified. It is initialized at **false**. It then sequentially goes through all the input controls. If any of the values displayed in the controls disagree with the stored values, the user has changed something and **edited** is set to **true**. If **edited** is **true**, the user is asked if they would like to save the entry.

Make the specified changes to your project. Save and run the project. Add a new inventory item, make some entries describing the item and click **Save**. Now modify something about your new item and click **New**, **Previous** or **Next**. You should see this message box:



You decide whether to save the changes or not. If you do, notice you don't have to click the **Save** button; it is done for you in code.

Suven Internship

Code Design – Inventory Item Search

When a user clicks one of the search button controls (**searchButton** array), the following happens:

- Determine which search button was clicked.
- Find first item in inventory list that begins with 'clicked' letter – display that item.
- If no matching item found, display a message box.

The code to implement these steps is straightforward and is placed in the **searchLabelActionPerformed** method:

```
private void searchButtonActionPerformed(ActionEvent e)
{
    int i;
    if (numberEntries == 0)
        return;
    // search for item letter
    String letterClicked = e.getActionCommand();
    i = 0;
    do
    {
        if (myInventory[i].description.substring(0, 1).equals(letterClicked))
        {
            currentEntry = i + 1;
            showEntry(currentEntry);
            return;
        }
        i++;
    }
    while (i < numberEntries);
    JOptionPane.showConfirmDialog(null, "No " + letterClicked + " inventory items.",
    "None Found", JOptionPane.DEFAULT_OPTION,
    JOptionPane.INFORMATION_MESSAGE);
}
```

Add this method to your project.

Save and run the project. Notice how the search labels are properly created and positioned. When I

click the 'T' search label using the sample inventory, I see:

New

Delete

Save

Previous

Next

Print

Exit

Inventory Item

TIVO

Location

Family Room

Serial Number

T4979387D

Purchase Price

199.00

Date Purchased

Jul 15, 2004

Store/Website

Best Buy


Note

Photo

C:\HomeJava\HomeJava Projects\Inventory Photos\tivo.jpg

Item Search

A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R
S	T	U	V	W	X
Y	Z				



Notice we see the first T entry (TIVO). To see entries 'around' this choice, use the Previous and Next buttons. Click on 'R' and you'll see:

None Found

No R inventory items.

OK

Printing with Java

One last capability we will add to our home inventory project is printing. A printed copy of our items would be helpful for archival purposes and for any potential insurance claims. Printing is one of the more tedious programming tasks within Java. But, fortunately, it is straightforward and there are dialog controls that help with the tasks. We will introduce lots of new topics here. All steps will be reviewed.

To perform printing in Java, we need this import statement:

```
import java.awt.print.*;
```

The **PrinterJob** class from this imported package controls the printing process. This class is used to start or cancel a printing job. It can also be used to display dialog boxes when needed. The **Printable** interface from this package is used to represent the item (document) to be printed.

The steps to print a document (which may include text and graphics) using the **PrinterJob** class are:

- Declare and create a **PrinterJob** object.
- Point the **PrinterJob** object to a **Printable** class (containing code to print the desired document) using the **setPrintable** method of the **PrinterJob** object.
- Print the document using the **print** method of the **PrinterJob** object.

These steps are straightforward. To declare and create a **PrinterJob** object named **myPrinterJob**, use:

```
PrinterJob myPrinterJob = PrinterJob.getPrinterJob();
```

If the **Printable** class is named **MyDocument**, the **PrinterJob** is associated with this class using:

```
myPrinterJob.setPrintable(new MyDocument());
```

Once associated, the printing is accomplished using the **print** method:

```
myPrinterJob.print();
```

This print method must be enclosed in a **try/catch** (catching a **PrinterException**) block.

The key to printing is the establishment of the **Printable** interface, called **MyDocument** here. This class describes the document to be printed and is placed after the main class. The form of this class is:

```
class MyDocument implements Printable
```

```
{  
    public int print(Graphics g, PageFormat pf, int pageIndex)  
    {  
        Graphics2D g2D = (Graphics2D) g;  
        .  
        .  
        .  
    }  
}
```

This class has a single method, **print**, which is called whenever the **PrinterJob** object needs information to do its job. In this method, you ‘construct’ each page (using Java code) that is to be printed. You’ll see the code in this method is familiar.

Note the **print** method has three arguments. The first argument is a **Graphics** object **g**. *Something familiar!* The **Printable** interface provides us with a graphics object to ‘draw’ each page we want to print. The second argument is a **PageFormat** object **pf**, which describes the size and orientation of the paper being used. Finally, the **pageIndex** argument is the number of the page to print. This argument is zero-based, meaning the first page has a value of zero.

The **print** method can return one of two constant values:

PAGE_EXISTS	returned if pageIndex refers to an existing page
NO_SUCH_PAGE	returned if pageIndex refers to a non-existing page

It is very important that **NO_SUCH_PAGE** is returned at some point or your program will assume there are an infinite number of pages to print!!

Another important thing to remember is that the **print** method may be called more than once per printed page, as the output is buffered to the printer. So, don’t build in any assumptions about how often **print** is called for a given page.

Summarizing the printing steps, here is basic Java code (**PrintingExample**) to print a document described by a class **MyDocument**:

```
import javax.swing.*;  
import java.awt.*;
```

```
import java.awt.print.*;
```

```
public class PrintingExample
```

```
{  
    public static void main(String[] args)  
    {
```

```
        PrinterJob myPrinterJob = PrinterJob.getPrinterJob();
```

```
        myPrinterJob.setPrintable(new MyDocument());
```

```
        try
```

```
        {
```

```
            myPrinterJob.print();
```

```
        }
```

```
        catch (PrinterException ex)
```

```
        {
```

```
            JOptionPane.showConfirmDialog(null, ex.getMessage(), "Print Error",  
JOptionPane.DEFAULT_OPTION, JOptionPane.ERROR_MESSAGE);
```

```
        }
```

```
    }
```

```
}
```

```
class MyDocument implements Printable
```

```
{
```

```
    public int print(Graphics g, PageFormat pf, int pageIndex)
```

```
    {
```

```
        Graphics2D g2D = (Graphics2D) g;
```

```
        .
```

```
        .
```

```
        .
```

```
    }
```

```
}
```

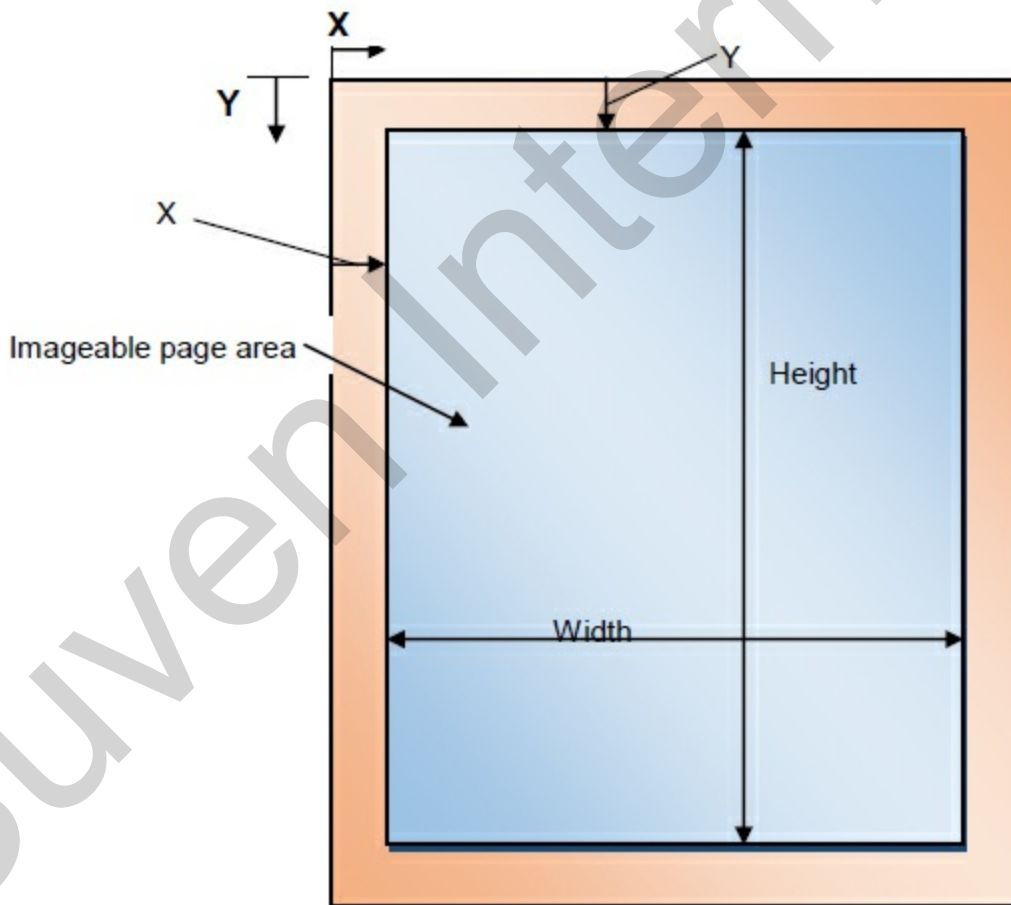
Let's see how to develop code for the **Printable** interface **print** method to do some printing.

Printing Document Pages

The **Printable** interface provides (in its **print** method) a graphics object (**g**, which we cast to a **Graphics2D** object, **g2D**) for ‘drawing’ our pages. And, that’s just what we do using familiar graphics methods. For each page in our printed document, we draw the desired text information (**drawString** method), any shapes (**draw** method), or graphics (using the **drawImage** we used to ‘draw’ the inventory photo).

Once a page is completely drawn to the graphics object, we ‘tell’ the **PrinterJob** object to print it. We repeat this process for each page we want to print. As noted, the **pageIndex** argument (in conjunction with the **return** value) of the print method helps with this effort. This does require a little bit of work on your part. You must know how many pages your document has and what goes on each page.

Let’s look at the coordinates and dimensions of the graphics object for a single page.



This becomes our palette for positioning items on a page. Horizontal position is governed by **X** (increases from 0 to the right) and vertical position is governed by **Y** (increases from 0 to the bottom). All dimensions are type **double**, in units of 1/72 inch. A standard sheet of 8.5 inch by 11-inch paper (with zero margins) would have a width and height of 612 and 792, respectively.

The **imageable area** rectangle is described by the **PageFormat** argument (**pf**) of the **Printable** class

print method. The origin can be determined using:

```
pf.getImageableX();
```

```
pf.getImageableY();
```

These values define the right and top margins, respectively. The width and height of the imageable area, respectively, are found using:

```
pf.getImageableWidth();
```

```
pf.getImageableHeight();
```

The returned values are **double** types in units of 1/72 inch.

The process for each page is to decide “what goes where” and then position the desired information using the appropriate graphics method. Any of the graphics methods we have learned can be used to put information on the graphic object.

To place text on the graphics object (**g2D**), use the **drawString** method. To place the string **myString** at position (**x**, **y**), the syntax is:

```
g2D.drawString(myString, x, y);
```

The string is printed using the current font and paint attributes. With this statement, you can place any text, anywhere you like, with any font and paint. You just need to make the desired specifications. Each line of text on a printed page will require a **drawString** statement. Note **x** and **y** in this method are **int** types, not **double**, hence type casting of page dimensions is usually needed.

Determine the width and height of strings (knowing the font object **myFont**). This is helpful for both vertical and horizontal placement of text on a page. This information is returned in a **Rectangle2D** structure (**stringRect**), using:

```
Rectangle2D stringRect = myFont.getStringBounds(myString,  
g2D.getFontRenderContext());
```

The height and width of the returned **stringRect** structure yield the string size (in units of 1/72 inch). These two properties are useful for justifying (left, right, center, vertical) text strings.

Many times, you use lines in a document to delineate various sections. To draw a line on the graphics object, use the **draw** method and **Line2D** shape :

```
Line2D.Double myLine = new Line2D.Double(x1, y1, x2, y2);  
g2D.draw(myLine);
```


This statement will draw a line from (x1, y1) to (x2, y2) using the current **stroke** and **paint** attributes.

Finally, the **drawImage** method is used to position an image (**myImage**) object on a page. This is the same method we used earlier to place the inventory photo on **photoPanel**. The syntax is:

```
g2D.drawImage(myImage, x, y, width, height,null);
```

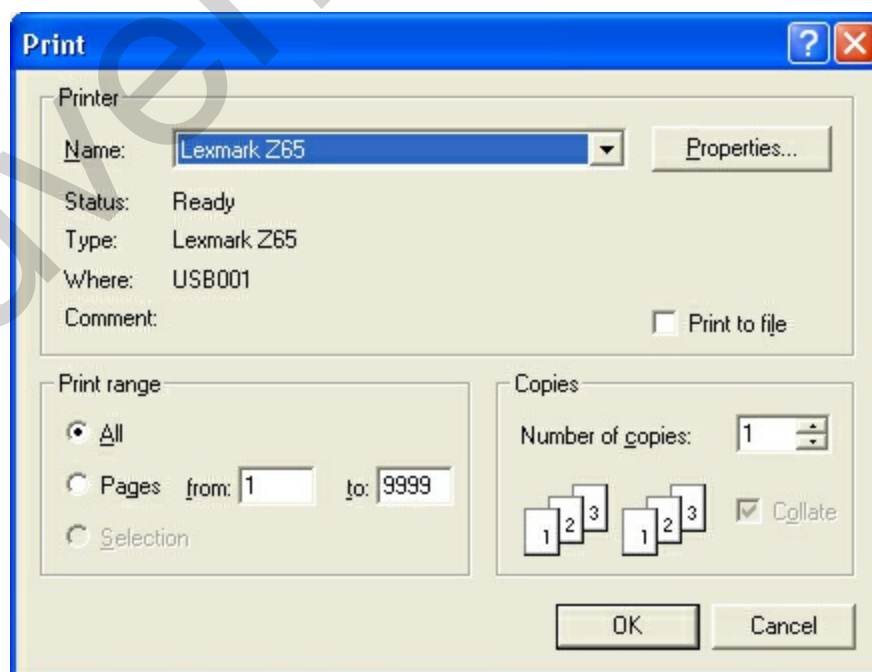
The upper left corner of **myImage** will be at (x, y) with the specified **width** and **height**. Any image will be scaled to fit the specified region.

We've seen all of these graphics methods before, so their use should be familiar. You should note that each item on a printed page requires at least one line of code. That results in lots of coding for printing. So, if you're writing lots of code in your print routines, you're probably doing it right.

Recall when doing persistent graphics using a **paintComponent** method in another class, any variable needed by that method needed to be prefaced by the keyword **static**. That is also needed here. Any class level object referred to in the **print** method of the **Printable** class must also be declared with a **static** preface.

Many print jobs just involve the user clicking a button marked '**Print**' and the results appear on the printed page with no further interaction. If more interaction is desired, there is a methods associated with the **PrinterJob** class that helps specify desired printing job properties: **printDialog**.

The **printDialog** method displays a dialog box that allows the user to select which printer to use, choose page orientation, printed page range and number of copies. This is the same dialog box that appears in many applications. The Windows version of the print dialog is:



The **printDialog** method returns **true** if the user clicked the **OK** button to leave the dialog and **false**

otherwise. After the method returns a value, you don't have to do anything to retrieve the parameters the user selected. The **PrinterJob** object is automatically updated with the selections!

Suven Internship

Code Design – Printing the Inventory

The format used for the printed inventory is straightforward – modify it as you see fit. Each page will have a simple header (giving the page number) and will hold two items from the inventory. Each property for each item (including the picture) will be printed. Items will be separated by a single straight line.

The code to establish the print document and display the printed inventory goes in the **printButtonActionPerformed** method. But, first, we need two class (**HomeInventory** class) level variable declarations to keep track of the printing process:

```
static final int entriesPerPage = 2;  
static int lastPage;
```

They are prefaced with **static** because they will be used in the **PrintJob** class. Also add the **static** preface to **numberEntries**, **myInventory** and **photoTextArea** since we will need these to print everything:

```
static int numberEntries;  
static InventoryItem[] myInventory = new InventoryItem[maximumEntries];  
static JTextArea photoTextArea = new JTextArea();
```

Lastly, make sure you have added this import statement:

```
import java.awt.print.*;
```

The **printButtonActionPerformed** method is then:

```
private void printButtonActionPerformed(ActionEvent e)  
{  
    lastPage = (int) (1 + (numberEntries - 1) / entriesPerPage);  
  
    PrinterJob inventoryPrinterJob = PrinterJob.getPrinterJob();  
    inventoryPrinterJob.setPrintable(new InventoryDocument());  
    if (inventoryPrinterJob.printDialog())  
    {  
        try  
        {  
            inventoryPrinterJob.print();  
        }  
    }  
}
```

```

        catch (PrinterException ex)
        {
            JOptionPane.showConfirmDialog(null, ex.getMessage(), "Print Error",
            JOptionPane.DEFAULT_OPTION, JOptionPane.ERROR_MESSAGE);
        }
    }
}

```

We determine the **lastPage** to print and create the **PrinterJob** object (**inventoryPrinterJob**). This object 'points' to the **Printable** class, **InventoryDocument**. A print dialog is displayed. If the user chooses **OK**, the **print** method is called. Add this method and the variable declarations to your project.

The **Printable** class to accomplish the printing is (place this after the **HomeInventory** and **PhotoPanel** classes)::

class InventoryDocument implements Printable

```

{
    public int print(Graphics g, PageFormat pf, int pageIndex)
    {
        Graphics2D g2D = (Graphics2D) g;
        if ((pageIndex + 1) > HomeInventory.lastPage)
        {
            return NO_SUCH_PAGE;
        }
        int i, iEnd;
        // here you decide what goes on each page and draw it
        // header
        g2D.setFont(new Font("Arial", Font.BOLD, 14));
        g2D.drawString("Home Inventory Items - Page " + String.valueOf(pageIndex + 1),
        (int) pf.getImageableX(), (int) (pf.getImageableY() + 25));
        // get starting y
        int dy = (int) g2D.getFont().getStringBounds("S",
        g2D.getFontRenderContext()).getHeight();
        int y = (int) (pf.getImageableY() + 4 * dy);
        iEnd = HomeInventory.entriesPerPage * (pageIndex + 1);
        if (iEnd > HomeInventory.numberEntries)
            iEnd = HomeInventory.numberEntries;
    }
}

```

```
for (i = 0 + HomeInventory.entriesPerPage * pageIndex; i < iEnd; i++)
```

```
{
```

```
    // dividing line
```

```
    Line2D.Double dividingLine = new Line2D.Double(pf.getImageableX(), y,  
pf.getImageableX() + pf.getImageableWidth(), y);
```

```
    g2D.draw(dividingLine);
```

```
    y += dy;
```

```
    g2D.setFont(new Font("Arial", Font.BOLD, 12));
```

```
g2D.drawString(HomeInventory.myInventory[i].description, (int) pf.getImageableX(), y);
```

```
    y += dy;
```

```
    g2D.setFont(new Font("Arial", Font.PLAIN, 12));
```

```
    g2D.drawString("Location: " + HomeInventory.myInventory[i].location, (int)  
(pf.getImageableX() + 25), y);
```

```
    y += dy;
```

```
    if (HomeInventory.myInventory[i].marked)
```

```
        g2D.drawString("Item is marked with identifying information.", (int)  
(pf.getImageableX() + 25), y);
```

```
    else
```

```
        g2D.drawString("Item is NOT marked with identifying information.", (int)  
(pf.getImageableX() + 25), y);
```

```
    y += dy;
```

```
    g2D.drawString("Serial Number: " +
```

```
HomeInventory.myInventory[i].serialNumber, (int) (pf.getImageableX() + 25), y);
```

```
    y += dy;
```

```
    g2D.drawString("Price: $" + HomeInventory.myInventory[i].purchasePrice + "  
Purchased on: " + HomeInventory.myInventory[i].purchaseDate, (int) (pf.getImageableX() +  
25), y);
```

```
    y += dy;
```

```
    g2D.drawString("Purchased at: " +  
HomeInventory.myInventory[i].purchaseLocation, (int) (pf.getImageableX() + 25), y);
```

```
    y += dy;
```

```
    g2D.drawString("Note: " + HomeInventory.myInventory[i].note, (int)  
(pf.getImageableX() + 25), y);
```

```
    y += dy;
```

```
    try
```

```
    {
```

```
        // maintain original width/height ratio
```

```

        Image inventoryImage = new
ImageIcon(HomeInventory.myInventory[i].photoFile).getImage ();
        double ratio = (double) (inventoryImage.getWidth(null)) / (double)
inventoryImage.getHeight(null);
        g2D.drawImage(inventoryImage, (int) (pf.getImageableX() + 25), y, (int) (100 *
ratio), 100, null);
    }
    catch (Exception ex)
    {
        // have place to go in case image file doesn't open
    }
    y += 2 * dy + 100;
}
return PAGE_EXISTS;
}
}

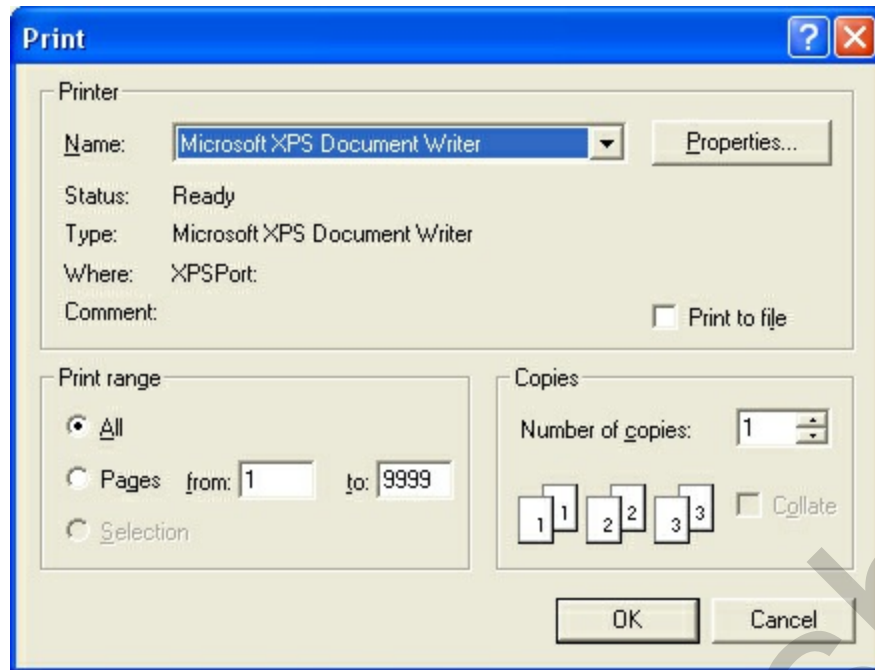
```

Yes, there's lots of code here, but the steps are straightforward:

- Print the header.
- For next two items in inventory:
 - o Draw dividing line.
 - o Print on separate lines: **Description, Location, Marked Statement, Serial Number, Purchase Price, Purchase Date, Purchase Location**, and any **Note**.
 - o Print picture (maintaining original height-to-width ratio).
- Check to see if more pages are to be printed.

You should see all of these steps in the above code. Note specifically how the vertical print location is updated using the text height.

Save, run the project one last time. Click the **Print** button on the toolbar. For the example inventory, the print dialog should appear:



You can choose how many (if not all) pages to print here and click OK, if your wish. A nicely formatted printing of your inventory will be obtained.

Home Inventory Manager Project Review

The **Home Inventory Manager Project** is now complete. Save and run the project and make sure it works as designed. Use the program to keep track of your belongings. You'll want to delete the **inventory.txt** file currently in your project folder and start over adding your own items and establishing your own combo box elements.

If there are errors in your implementation, go back over the steps of frame and code design. Go over the developed code – make sure you understand how different parts of the project were coded.

While completing this project, new concepts and skills you should have gained include:

- Use of combo box control.
- Basic object-oriented programming concepts and how to define your own classes and objects.
- How to add printing to a project, including use of the print dialog control.

Home Inventory Manager Project Enhancements

Possible enhancements to the home inventory manager project include:

- After clicking the **New** toolbar button, you must add an item to the inventory and click **Save**. There is no **Cancel** option – add such an option.
- The implemented search is rather basic. Add a search capability that looks through all the information in the inventory for certain terms or parts of terms.
- Modify the project to allow opening and saving of separate inventory files. That is, replace the built-in file (inventory.txt) with one you open/save using the dialog controls.

Home Inventory Manager Project Java Code Listing

There are two files, **HomeInventory.java** and **InventoryItem**.

HomeInventory.java:

```
/*
 * HomeInventory.java
 */

package homeinventory;
import javax.swing.*;
import javax.swing.filechooser.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import com.toedter.calendar.*;
import java.awt.geom.*;
import java.io.*;
import java.util.*;
import java.text.*;
import java.awt.print.*;

public class HomeInventory extends JFrame
{

    // Toolbar
    JToolBar inventoryToolBar = new JToolBar();
    JButton newButton = new JButton(new ImageIcon("new.gif"));
    JButton deleteButton = new JButton(new ImageIcon("delete.gif"));
    JButton saveButton = new JButton(new ImageIcon("save.gif"));
    JButton previousButton = new JButton(new ImageIcon("previous.gif"));
    JButton nextButton = new JButton(new ImageIcon("next.gif"));
    JButton printButton = new JButton(new ImageIcon("print.gif"));
    JButton exitButton = new JButton();
```

// Frame

JLabel itemLabel = new JLabel();

JTextField itemTextField = new JTextField();

JLabel locationLabel = new JLabel();

JComboBox locationComboBox = new JComboBox();

JCheckBox markedCheckBox = new JCheckBox();

JLabel serialLabel = new JLabel();

JTextField serialTextField = new JTextField();

JLabel priceLabel = new JLabel();

JTextField priceTextField = new JTextField();

JLabel dateLabel = new JLabel();

JDateChooser dateDateChooser = new JDateChooser();

JLabel storeLabel = new JLabel();

JTextField storeTextField = new JTextField();

JLabel noteLabel = new JLabel();

JTextField noteTextField = new JTextField();

JLabel photoLabel = new JLabel();

static JTextArea photoTextArea = new JTextArea();

JButton photoButton = new JButton();

JPanel searchPanel = new JPanel();

JButton[] searchButton = new JButton[26];

PhotoPanel photoPanel = new PhotoPanel();

static final int maximumEntries = 300;

static int numberEntries;

static InventoryItem[] myInventory = new InventoryItem[maximumEntries];

int currentEntry;

static final int entriesPerPage = 2;

static int lastPage;

public static void main(String args[])

{

// create frame

new HomeInventory().show();

}

public HomeInventory()

{

// frame constructor

setTitle("Home Inventory Manager");

setResizable(false);

setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

addWindowListener(new WindowAdapter()

{

public void windowClosing(WindowEvent evt)

{

exitForm(evt);

}

});

getContentPane().setLayout(new GridBagLayout());

GridBagConstraints gridConstraints;

inventoryToolBar.setFloatable(false);

inventoryToolBar.setBackground(Color.BLUE);

inventoryToolBar.setOrientation(SwingConstants.VERTICAL);

gridConstraints = new GridBagConstraints();

gridConstraints.gridx = 0;

gridConstraints.gridy = 0;

gridConstraints.gridheight = 8;

gridConstraints.fill = GridBagConstraints.VERTICAL;

getContentPane().add(inventoryToolBar, gridConstraints);

inventoryToolBar.addSeparator();

Dimension bSize = new Dimension(70, 50);

newButton.setText("New");

sizeButton(newButton, bSize);

newButton.setToolTipText("Add New Item");

newButton.setHorizontalTextPosition(SwingConstants.CENTER);

newButton.setVerticalTextPosition(SwingConstants.BOTTOM);

newButton.setFocusable(false);

inventoryToolBar.add(newButton);

```
newButton.addActionListener(new ActionListener()  
{  
    public void actionPerformed(ActionEvent e)  
    {  
        newButtonActionPerformed(e);  
    }  
});
```

```
deleteButton.setText("Delete");  
sizeButton(deleteButton, bSize);  
deleteButton.setToolTipText("Delete Current Item");
```

```
deleteButton.setHorizontalTextPosition(SwingConstants.CENTER);
```

```
deleteButton.setVerticalTextPosition(SwingConstants.BOTTOM);  
deleteButton.setFocusable(false);  
inventoryToolBar.add(deleteButton);  
deleteButton.addActionListener(new ActionListener()  
{  
    public void actionPerformed(ActionEvent e)  
    {  
        deleteButtonActionPerformed(e);  
    }  
});
```

```
saveButton.setText("Save");  
sizeButton(saveButton, bSize);  
saveButton.setToolTipText("Save Current Item");
```

```
saveButton.setHorizontalTextPosition(SwingConstants.CENTER);
```

```
saveButton.setVerticalTextPosition(SwingConstants.BOTTOM);  
saveButton.setFocusable(false);  
inventoryToolBar.add(saveButton);  
saveButton.addActionListener(new ActionListener()  
{  
    public void actionPerformed(ActionEvent e)  
    {
```

```
        saveButtonActionPerformed(e);
    }
});

inventoryToolBar.addSeparator();

previousButton.setText("Previous");
sizeButton(previousButton, bSize);
previousButton.setToolTipText("Display Previous Item");

previousButton.setHorizontalTextPosition(SwingConstants.CENTER);

previousButton.setVerticalTextPosition(SwingConstants.BOTTOM);
previousButton.setFocusable(false);
inventoryToolBar.add(previousButton);
previousButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        previousButtonActionPerformed(e);
    }
});

nextButton.setText("Next");
sizeButton(nextButton, bSize);
nextButton.setToolTipText("Display Next Item");

nextButton.setHorizontalTextPosition(SwingConstants.CENTER);

nextButton.setVerticalTextPosition(SwingConstants.BOTTOM);
nextButton.setFocusable(false);
inventoryToolBar.add(nextButton);
nextButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        nextButtonActionPerformed(e);
    }
});
```

});

inventoryToolBar.addSeparator();

printButton.setText("Print");

sizeButton(printButton, bSize);

printButton.setToolTipText("Print Inventory List");

printButton.setHorizontalTextPosition(SwingConstants.CENTER);

printButton.setVerticalTextPosition(SwingConstants.BOTTOM);

printButton.setFocusable(false);

inventoryToolBar.add(printButton);

printButton.addActionListener(new ActionListener()

{

public void actionPerformed(ActionEvent e)

{

printButtonActionPerformed(e);

}

});

exitButton.setText("Exit");

sizeButton(exitButton, bSize);

exitButton.setToolTipText("Exit Program");

exitButton.setFocusable(false);

inventoryToolBar.add(exitButton);

exitButton.addActionListener(new ActionListener()

{

public void actionPerformed(ActionEvent e)

{

exitButtonActionPerformed(e);

}

});

itemLabel.setText("Inventory Item");

gridConstraints = new GridBagConstraints();

gridConstraints.gridx = 1;

gridConstraints.gridy = 0;

```
gridConstraints.insets = new Insets(10, 10, 0, 10);
gridConstraints.anchor = GridBagConstraints.EAST;
getContentPane().add(itemLabel, gridConstraints);

itemTextField.setPreferredSize(new Dimension(400, 25));
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 2;
gridConstraints.gridy = 0;
gridConstraints.gridwidth = 5;
gridConstraints.insets = new Insets(10, 0, 0, 10);
gridConstraints.anchor = GridBagConstraints.WEST;
getContentPane().add(itemTextField, gridConstraints);
itemTextField.addActionListener(new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {
        itemTextFieldActionPerformed(e);
    }
});
```

```
locationLabel.setText("Location");
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 1;
gridConstraints.gridy = 1;
gridConstraints.insets = new Insets(10, 10, 0, 10);
gridConstraints.anchor = GridBagConstraints.EAST;
getContentPane().add(locationLabel, gridConstraints);
```

```
locationComboBox.setPreferredSize(new Dimension(270, 25));
locationComboBox.setFont(new Font("Arial", Font.PLAIN, 12));
locationComboBox.setEditable(true);
locationComboBox.setBackground(Color.WHITE);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 2;
gridConstraints.gridy = 1;
gridConstraints.gridwidth = 3;
gridConstraints.insets = new Insets(10, 0, 0, 10);
```



```
gridConstraints.anchor = GridBagConstraints.WEST;
getContentPane().add(locationComboBox, gridConstraints);
locationComboBox.addActionListener(new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {
        locationComboBoxActionPerformed(e);
    }
});
```

```
markedCheckBox.setText("Marked?");
markedCheckBox.setFocusable(false);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 5;
gridConstraints.gridy = 1;
gridConstraints.insets = new Insets(10, 10, 0, 0);
gridConstraints.anchor = GridBagConstraints.WEST;
getContentPane().add(markedCheckBox, gridConstraints);
```

```
serialLabel.setText("Serial Number");
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 1;
gridConstraints.gridy = 2;
gridConstraints.insets = new Insets(10, 10, 0, 10);
gridConstraints.anchor = GridBagConstraints.EAST;
getContentPane().add(serialLabel, gridConstraints);
```

```
serialTextField.setPreferredSize(new Dimension(270, 25));
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 2;
gridConstraints.gridy = 2;
gridConstraints.gridwidth = 3;
gridConstraints.insets = new Insets(10, 0, 0, 10);
gridConstraints.anchor = GridBagConstraints.WEST;
getContentPane().add(serialTextField, gridConstraints);
serialTextField.addActionListener(new ActionListener ()
{
```

```

    public void actionPerformed(ActionEvent e)
    {
        serialTextFieldActionPerformed(e);
    }
});

priceLabel.setText("Purchase Price");
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 1;
gridConstraints.gridy = 3;
gridConstraints.insets = new Insets(10, 10, 0, 10);
gridConstraints.anchor = GridBagConstraints.EAST;
getContentPane().add(priceLabel, gridConstraints);
priceTextField.setPreferredSize(new Dimension(160, 25));
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 2;
gridConstraints.gridy = 3;
gridConstraints.gridwidth = 2;
gridConstraints.insets = new Insets(10, 0, 0, 10);
gridConstraints.anchor = GridBagConstraints.WEST;
getContentPane().add(priceTextField, gridConstraints);
priceTextField.addActionListener(new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {
        priceTextFieldActionPerformed(e);
    }
});

dateLabel.setText("Date Purchased");
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 4;
gridConstraints.gridy = 3;
gridConstraints.insets = new Insets(10, 10, 0, 0);
gridConstraints.anchor = GridBagConstraints.WEST;
getContentPane().add(dateLabel, gridConstraints);

```

```
dateDateChooser.setPreferredSize(new Dimension(120, 25));
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 5;
gridConstraints.gridy = 3;
gridConstraints.gridwidth = 2;
gridConstraints.insets = new Insets(10, 0, 0, 10);
gridConstraints.anchor = GridBagConstraints.WEST;
getContentPane().add(dateDateChooser, gridConstraints);
dateDateChooser.addPropertyChangeListener(new PropertyChangeListener()
{
    public void propertyChange(PropertyChangeEvent e)
    {
        dateDateChooserPropertyChange(e);
    }
});
```

```
storeLabel.setText("Store/Website");
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 1;
gridConstraints.gridy = 4;
gridConstraints.insets = new Insets(10, 10, 0, 10);
gridConstraints.anchor = GridBagConstraints.EAST;
getContentPane().add(storeLabel, gridConstraints);
```

```
storeTextField.setPreferredSize(new Dimension(400, 25));
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 2;
gridConstraints.gridy = 4;
gridConstraints.gridwidth = 5;
gridConstraints.insets = new Insets(10, 0, 0, 10);
gridConstraints.anchor = GridBagConstraints.WEST;
getContentPane().add(storeTextField, gridConstraints);
storeTextField.addActionListener(new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {
```

```

        storeTextFieldActionPerformed(e);
    }
});

noteLabel.setText("Note");
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 1;
gridConstraints.gridy = 5;
gridConstraints.insets = new Insets(10, 10, 0, 10);
gridConstraints.anchor = GridBagConstraints.EAST;
getContentPane().add(noteLabel, gridConstraints);

noteTextField.setPreferredSize(new Dimension(400, 25));
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 2;
gridConstraints.gridy = 5;
gridConstraints.gridwidth = 5;
gridConstraints.insets = new Insets(10, 0, 0, 10);
gridConstraints.anchor = GridBagConstraints.WEST;
getContentPane().add(noteTextField, gridConstraints);
noteTextField.addActionListener(new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {
        noteTextFieldActionPerformed(e);
    }
});

photoLabel.setText("Photo");
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 1;
gridConstraints.gridy = 6;
gridConstraints.insets = new Insets(10, 10, 0, 10);
gridConstraints.anchor = GridBagConstraints.EAST;
getContentPane().add(photoLabel, gridConstraints);

photoTextArea.setPreferredSize(new Dimension(350, 35));

```

```
photoTextArea.setFont(new Font("Arial", Font.PLAIN, 12));
photoTextArea.setEditable(false);
photoTextArea.setLineWrap(true);
photoTextArea.setWrapStyleWord(true);
photoTextArea.setBackground(new Color(255, 255, 192));
```

```
photoTextArea.setBorder(BorderFactory.createLineBorder(Color.BLACK));
photoTextArea.setFocusable(false);
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 2;
gridConstraints.gridy = 6;
gridConstraints.gridwidth = 4;
gridConstraints.insets = new Insets(10, 0, 0, 10);
gridConstraints.anchor = GridBagConstraints.WEST;
getContentPane().add(photoTextArea, gridConstraints);
```

```
photoButton.setText("...");
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 6;
gridConstraints.gridy = 6;
gridConstraints.insets = new Insets(10, 0, 0, 10);
gridConstraints.anchor = GridBagConstraints.WEST;
getContentPane().add(photoButton, gridConstraints);
photoButton.addActionListener(new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {
        photoButtonActionPerformed(e);
    }
});
```

```
searchPanel.setPreferredSize(new Dimension(240, 160));
```

```
searchPanel.setBorder(BorderFactory.createTitledBorder("Item Search"));
searchPanel.setLayout(new GridBagLayout());
gridConstraints = new GridBagConstraints();
gridConstraints.gridx = 1;
```

```
gridConstraints.gridy = 7;  
gridConstraints.gridwidth = 3;  
gridConstraints.insets = new Insets(10, 0, 10, 0);  
gridConstraints.anchor = GridBagConstraints.CENTER;  
getContentPane().add(searchPanel, gridConstraints);
```

```
int x = 0, y = 0;
```

```
// create and position 26 buttons
```

```
for (int i = 0; i < 26; i++)
```

```
{
```

```
    // create new button
```

```
    searchButton[i] = new JButton();
```

```
    // set text property
```

```
    searchButton[i].setText(String.valueOf((char) (65 + i)));
```

```
    searchButton[i].setFont(new Font("Arial", Font.BOLD, 12));
```

```
    searchButton[i].setMargin(new Insets(-10, -10, -10, -10));
```

```
    sizeButton(searchButton[i], new Dimension(37, 27));
```

```
    searchButton[i].setBackground(Color.YELLOW);
```

```
    searchButton[i].setFocusable(false);
```

```
    gridConstraints = new GridBagConstraints();
```

```
    gridConstraints.gridx = x;
```

```
    gridConstraints.gridy = y;
```

```
    searchPanel.add(searchButton[i], gridConstraints);
```

```
    // add method
```

```
    searchButton[i].addActionListener(new ActionListener ()
```

```
    {
```

```
        public void actionPerformed(ActionEvent e)
```

```
        {
```

```
            searchButtonActionPerformed(e);
```

```
        }
```

```
    });
```

```
    x++;
```

```
// six buttons per row
```

```
if (x % 6 == 0)
```

```
{
```

```
    x = 0;
```

```
        y++;  
    }  
}
```

```
photoPanel.setPreferredSize(new Dimension(240, 160));  
gridConstraints = new GridBagConstraints();  
gridConstraints.gridx = 4;  
gridConstraints.gridy = 7;  
gridConstraints.gridwidth = 3;  
gridConstraints.insets = new Insets(10, 0, 10, 10);  
gridConstraints.anchor = GridBagConstraints.CENTER;  
getContentPane().add(photoPanel, gridConstraints);
```

```
pack();
```

```
Dimension screenSize =
```

```
Toolkit.getDefaultToolkit().getScreenSize();
```

```
setBounds((int) (0.5 * (screenSize.width - getWidth())), (int) (0.5 * (screenSize.height -  
getHeight())), getWidth(), getHeight());
```

```
int n;
```

```
// open file for entries
```

```
try
```

```
{
```

```
    BufferedReader inputFile = new BufferedReader(new FileReader("inventory.txt"));
```

```
    numberEntries =
```

```
Integer.valueOf(inputFile.readLine()).intValue();
```

```
    if (numberEntries != 0)
```

```
    {
```

```
        for (int i = 0; i < numberEntries; i++)
```

```
        {
```

```
            myInventory[i] = new InventoryItem();
```

```
            myInventory[i].description = inputFile.readLine();
```

```
            myInventory[i].location = inputFile.readLine();
```

```
            myInventory[i].serialNumber = inputFile.readLine();
```

```
            myInventory[i].marked =
```

```
Boolean.valueOf(inputFile.readLine()).booleanValue();
```

```
            myInventory[i].purchasePrice =
```

```

inputFile.readLine();
        myInventory[i].purchaseDate = inputFile.readLine();
        myInventory[i].purchaseLocation =
inputFile.readLine();
        myInventory[i].note = inputFile.readLine();
        myInventory[i].photoFile = inputFile.readLine();
    }
}
// read in combo box elements
n = Integer.valueOf(inputFile.readLine()).intValue();
if (n != 0)
{
    for (int i = 0; i < n; i++)
    {
        locationComboBox.addItem(inputFile.readLine());
    }
}
inputFile.close();
currentEntry = 1;
showEntry(currentEntry);
}
catch (Exception ex)
{
    numberEntries = 0;
    currentEntry = 0;
}
if (numberEntries == 0)
{
    newButton.setEnabled(false);
    deleteButton.setEnabled(false);
    nextButton.setEnabled(false);
    previousButton.setEnabled(false);
    printButton.setEnabled(false);
}
}
}

```



```
private void exitForm(WindowEvent evt)
```

```
{
```

```
    if (JOptionPane.showConfirmDialog(null, "Any unsaved changes will be lost.\nAre you  
sure you want to exit?", "Exit Program", JOptionPane.YES_NO_OPTION,  
JOptionPane.QUESTION_MESSAGE) == JOptionPane.NO_OPTION)
```

```
        return;
```

```
    // write entries back to file
```

```
    try
```

```
    {
```

```
        PrintWriter outputFile = new PrintWriter(new BufferedWriter(new  
File Writer("inventory.txt")));
```

```
        outputFile.println(numberEntries);
```

```
        if (numberEntries != 0)
```

```
        {
```

```
            for (int i = 0; i < numberEntries; i++)
```

```
            {
```

```
                outputFile.println(myInventory[i].description);
```

```
                outputFile.println(myInventory[i].location);
```

```
                outputFile.println(myInventory[i].serialNumber);
```

```
                outputFile.println(myInventory[i].marked);
```

```
                outputFile.println(myInventory[i].purchasePrice);
```

```
                outputFile.println(myInventory[i].purchaseDate);
```

```
        outputFile.println(myInventory[i].purchaseLocation);
```

```
        outputFile.println(myInventory[i].note);
```

```
        outputFile.println(myInventory[i].photoFile);
```

```
    }
```

```
    }
```

```
    // write combo box entries
```

```
    outputFile.println(locationComboBox.getItemCount());
```

```
    if (locationComboBox.getItemCount() != 0)
```

```
    {
```

```
        for (int i = 0; i < locationComboBox.getItemCount(); i++)
```

```
            outputFile.println(locationComboBox.getItemAt(i));
```

```
    }
```

```
    outputFile.close();
```

```
}
```

```

    catch (Exception ex)
    {

    }

    System.exit(0);
}

private void newButtonActionPerformed(ActionEvent e)
{
    checkSave();
    blankValues();
}

private void deleteButtonActionPerformed(ActionEvent e)
{
    if (JOptionPane.showConfirmDialog(null, "Are you sure you want to delete this item?",
    "Delete Inventory Item", JOptionPane.YES_NO_OPTION,
    JOptionPane.QUESTION_MESSAGE) == JOptionPane.NO_OPTION)
        return;
    deleteEntry(currentEntry);
    if (numberEntries == 0)
    {
        currentEntry = 0;
        blankValues();
    }
    else
    {
        currentEntry--;
        if (currentEntry == 0)
            currentEntry = 1;
        showEntry(currentEntry);
    }
}

private void saveButtonActionPerformed(ActionEvent e)
{
    // check for description

```

```

itemTextField.setText(itemTextField.getText().trim());
if (itemTextField.getText().equals(""))
{
    JOptionPane.showConfirmDialog(null, "Must have item description.", "Error",
JOptionPane.DEFAULT_OPTION, JOptionPane.ERROR_MESSAGE);
    itemTextField.requestFocus();
    return;
}
if (newButton.isEnabled())
{
    // delete edit entry then resave
    deleteEntry(currentEntry);
}
// capitalize first letter
String s = itemTextField.getText();
itemTextField.setText(s.substring(0, 1).toUpperCase() + s.substring(1));
numberEntries++;
// determine new current entry location based on description
currentEntry = 1;
if (numberEntries != 1)
{
    do
    {
        if
(itemTextField.getText().compareTo(myInventory[currentEntry - 1].description) < 0)
            break;
        currentEntry++;
    }
    while (currentEntry < numberEntries);
}
// move all entries below new value down one position unless at end
if (currentEntry != numberEntries)
{
    for (int i = numberEntries; i >= currentEntry + 1; i--)
    {
        myInventory[i - 1] = myInventory[i - 2];
    }
}

```

```

        myInventory[i - 2] = new InventoryItem();
    }
}
myInventory[currentEntry - 1] = new InventoryItem();
myInventory[currentEntry - 1].description = itemTextField.getText();
myInventory[currentEntry - 1].location =
locationComboBox.getSelectedItem().toString();
myInventory[currentEntry - 1].marked = markedCheckBox.isSelected();
myInventory[currentEntry - 1].serialNumber = serialTextField.getText();
myInventory[currentEntry - 1].purchasePrice = priceTextField.getText();
myInventory[currentEntry - 1].purchaseDate =
dateToString(dateDateChooser.getDate());
myInventory[currentEntry - 1].purchaseLocation = storeTextField.getText();
myInventory[currentEntry - 1].photoFile = photoTextArea.getText();
myInventory[currentEntry - 1].note = noteTextField.getText();
showEntry(currentEntry);
if (numberEntries < maximumEntries)
    newButton.setEnabled(true);
else
    newButton.setEnabled(false);
deleteButton.setEnabled(true);
printButton.setEnabled(true);
}

private void previousButtonActionPerformed(ActionEvent e)
{
    checkSave();
    currentEntry--;
    showEntry(currentEntry);
}

private void nextButtonActionPerformed(ActionEvent e)
{
    checkSave();
    currentEntry++;
    showEntry(currentEntry);
}

```

```
}
```

```
private void printButtonActionPerformed(ActionEvent e)
```

```
{
```

```
    lastPage = (int) (1 + (numberEntries - 1) / entriesPerPage);
```

```
    PrinterJob inventoryPrinterJob = PrinterJob.getPrinterJob();
```

```
    inventoryPrinterJob.setPrintable(new InventoryDocument());
```

```
    if (inventoryPrinterJob.printDialog())
```

```
    {
```

```
        try
```

```
        {
```

```
            inventoryPrinterJob.print();
```

```
        }
```

```
        catch (PrinterException ex)
```

```
        {
```

```
            JOptionPane.showConfirmDialog(null, ex.getMessage(), "Print Error",
```

```
JOptionPane.DEFAULT_OPTION, JOptionPane.ERROR_MESSAGE);
```

```
        }
```

```
    }
```

```
}
```

```
private void exitButtonActionPerformed(ActionEvent e)
```

```
{
```

```
    exitForm(null);
```

```
}
```

```
private void photoButtonActionPerformed(ActionEvent e)
```

```
{
```

```
    JFileChooser openChooser = new JFileChooser();
```

```
    openChooser.setDialogType(JFileChooser.OPEN_DIALOG);
```

```
    openChooser.setDialogTitle("Open Photo File");
```

```
    openChooser.addChoosableFileFilter(new FileNameExtensionFilter("Photo Files",
```

```
"jpg"));
```

```
    if (openChooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION)
```

```
        showPhoto(openChooser.getSelectedFile().toString());
```

```
}
```

```
private void searchButtonActionPerformed(ActionEvent e)
```

```
{
```

```
    int i;
```

```
    if (numberEntries == 0)
```

```
        return;
```

```
    // search for item letter
```

```
    String letterClicked = e.getActionCommand();
```

```
    i = 0;
```

```
    do
```

```
    {
```

```
        if (myInventory[i].description.substring(0, 1).equals(letterClicked))
```

```
        {
```

```
            currentEntry = i + 1;
```

```
            showEntry(currentEntry);
```

```
            return;
```

```
        }
```

```
        i++;
```

```
    }
```

```
    while (i < numberEntries);
```

```
    JOptionPane.showConfirmDialog(null, "No " + letterClicked + " inventory items.",
```

```
"None Found", JOptionPane.DEFAULT_OPTION,
```

```
JOptionPane.INFORMATION_MESSAGE);
```

```
}
```

```
private void itemTextFieldActionPerformed(ActionEvent e)
```

```
{
```

```
    locationComboBox.requestFocus();
```

```
}
```

```
private void locationComboBoxActionPerformed(ActionEvent e)
```

```
{
```

```
    // If in list - exit method
```

```
    if (locationComboBox.getItemCount() != 0)
```

```
    {
```

```
        for (int i = 0; i < locationComboBox.getItemCount(); i++)
```

```
        {
```

```
            if (locationComboBox.getSelectedItem().toString().equals(locatio
```

```
nComboBox.getItemAt(i).toString()))
```

```
{
```

```
    serialTextField.requestFocus();
```

```
    return;
```

```
}
```

```
}
```

```
}
```

```
// If not found, add to list box
```

```
locationComboBox.addItem(locationComboBox.getSelectedItem());
```

```
    serialTextField.requestFocus();
```

```
}
```

```
private void serialTextFieldActionPerformed(ActionEvent e)
```

```
{
```

```
    priceTextField.requestFocus();
```

```
}
```

```
private void priceTextFieldActionPerformed(ActionEvent e)
```

```
{
```

```
    dateDateChooser.requestFocus();
```

```
}
```

```
private void dateDateChooserPropertyChange(PropertyChangeEvent e)
```

```
{
```

```
    storeTextField.requestFocus();
```

```
}
```

```
private void storeTextFieldActionPerformed(ActionEvent e)
```

```
{
```

```
    noteTextField.requestFocus();
```

```
}
```

```
private void noteTextFieldActionPerformed(ActionEvent e)
```

```
{
```

```
    photoButton.requestFocus();
```

```
}
```

```
private void sizeButton(JButton b, Dimension d)
```

```
{
```

```
    b.setPreferredSize(d);
```

```
    b.setMinimumSize(d);
```

```
    b.setMaximumSize(d);
```

```
}
```

```
private void showEntry(int j)
```

```
{
```

```
    // display entry j (1 to numberEntries)
```

```
    itemTextField.setText(myInventory[j - 1].description);
```

```
    locationComboBox.setSelectedItem(myInventory[j - 1].location);
```

```
    markedCheckBox.setSelected(myInventory[j - 1].marked);
```

```
    serialTextField.setText(myInventory[j - 1].serialNumber);
```

```
    priceTextField.setText(myInventory[j - 1].purchasePrice);
```

```
    dateDateChooser.setDate(stringToDate(myInventory[j - 1].purchaseDate));
```

```
    storeTextField.setText(myInventory[j - 1].purchaseLocation);
```

```
    noteTextField.setText(myInventory[j - 1].note);
```

```
    showPhoto(myInventory[j - 1].photoFile);
```

```
    nextButton.setEnabled(true);
```

```
    previousButton.setEnabled(true);
```

```
    if (j == 1)
```

```
        previousButton.setEnabled(false);
```

```
    if (j == numberEntries)
```

```
        nextButton.setEnabled(false);
```

```
    itemTextField.requestFocus();
```

```
}
```

```
private Date stringToDate(String s)
```

```
{
```

```
    int m = Integer.valueOf(s.substring(0, 2)).intValue() - 1;
```

```
    int d = Integer.valueOf(s.substring(3, 5)).intValue();
```

```
    int y = Integer.valueOf(s.substring(6)).intValue() - 1900;
```

```
    return(new Date(y, m, d));
```

```
}
```

```
private String dateToString(Date dd)
```



```
{  
    String yString = String.valueOf(dd.getYear() + 1900);  
    int m = dd.getMonth() + 1;  
    String mString = new DecimalFormat("00").format(m);  
    int d = dd.getDate();  
    String dString = new DecimalFormat("00").format(d);  
    return(mString + "/" + dString + "/" + yString);  
}
```

```
private void showPhoto(String photoFile)
```

```
{  
    if (!photoFile.equals(""))  
    {  
        try  
        {  
            photoTextArea.setText(photoFile);  
        }  
        catch (Exception ex)  
        {  
            photoTextArea.setText("");  
        }  
    }  
    else  
    {  
        photoTextArea.setText("");  
    }  
    photoPanel.repaint();  
}
```

```
private void blankValues()
```

```
{  
    // blank input screen  
    newButton.setEnabled(false);  
    deleteButton.setEnabled(false);  
    saveButton.setEnabled(true);  
    previousButton.setEnabled(false);  
}
```

```

nextButton.setEnabled(false);
printButton.setEnabled(false);
itemTextField.setText("");
locationComboBox.setSelectedItem("");
markedCheckBox.setSelected(false);
serialTextField.setText("");
priceTextField.setText("");
dateDateChooser.setDate(new Date());
storeTextField.setText("");
noteTextField.setText("");
photoTextArea.setText("");
photoPanel.repaint();
itemTextField.requestFocus();
}

private void deleteEntry(int j)
{
    // delete entry j
    if (j != numberEntries)
    {
        // move all entries under j up one level
        for (int i = j; i < numberEntries; i++)
        {
            myInventory[i - 1] = new InventoryItem();
            myInventory[i - 1] = myInventory[i];
        }
    }
    numberEntries--;
}

private void checkSave()
{
    boolean edited = false;
    if (!myInventory[currentEntry - 1].description.equals(itemTextField.getText()))
        edited = true;
    else if (!myInventory[currentEntry -

```

```

1].location.equals(locationComboBox.getSelectedItem().toString g()))
    edited = true;
else if (myInventory[currentEntry - 1].marked != markedCheckBox.isSelected())
    edited = true;
else if (!myInventory[currentEntry - 1].serialNumber.equals(serialTextField.getText()))
    edited = true;
else if (!myInventory[currentEntry - 1].purchasePrice.equals(priceTextField.getText()))
    edited = true;
else if (!myInventory[currentEntry -
1].purchaseDate.equals(dateToString(dateDateChooser.getDate()))
    edited = true;
else if (!myInventory[currentEntry -
1].purchaseLocation.equals(storeTextField.getText()))
    edited = true;
else if (!myInventory[currentEntry - 1].note.equals(noteTextField.getText()))
    edited = true;
else if (!myInventory[currentEntry - 1].photoFile.equals(photoTextArea.getText()))
    edited = true;
if (edited)
{
    if (JOptionPane.showConfirmDialog(null, "You have edited this item. Do you want to
save the changes?", "Save Item", JOptionPane.YES_NO_OPTION,
JOptionPane.QUESTION_MESSAGE) == JOptionPane.YES_OPTION)
        saveButton.doClick();
}
}
}

class PhotoPanel extends JPanel
{
    public void paintComponent(Graphics g)
    {
        Graphics2D g2D = (Graphics2D) g;
        super.paintComponent(g2D);

        // draw border
        g2D.setPaint(Color.BLACK);

```

```

    g2D.draw(new Rectangle2D.Double(0, 0, getWidth() - 1, getHeight() - 1));
    // show photo
    Image photoImage = new
ImageIcon(HomeInventory.photoTextArea.getText()).getImage();
    int w = getWidth();
    int h = getHeight();
    double rWidth = (double) getWidth() / (double) photoImage.getWidth(null);
    double rHeight = (double) getHeight() / (double) photoImage.getHeight(null);
    if (rWidth > rHeight)
    {
        // leave height at display height, change width by amount height is changed
        w = (int) (photoImage.getWidth(null) * rHeight);
    }
    else
    {
        // leave width at display width, change height by amount width is changed
        h = (int) (photoImage.getHeight(null) * rWidth);
    }
    // center in panel
    g2D.drawImage(photoImage, (int) (0.5 * (getWidth() - w)), (int) (0.5 * (getHeight() -
h)), w, h, null);

    g2D.dispose();
}
}

class InventoryDocument implements Printable
{
    public int print(Graphics g, PageFormat pf, int pageIndex)
    {
        Graphics2D g2D = (Graphics2D) g;
        if ((pageIndex + 1) > HomeInventory.lastPage)
        {
            return NO_SUCH_PAGE;
        }
        int i, iEnd;
        // here you decide what goes on each page and draw it

```

```

// header
g2D.setFont(new Font("Arial", Font.BOLD, 14));
g2D.drawString("Home Inventory Items - Page " + String.valueOf(pageIndex + 1),
(int) pf.getImageableX(), (int) (pf.getImageableY() + 25));
// get starting y
int dy = (int) g2D.getFont().getStringBounds("S",
g2D.getFontRenderContext()).getHeight();
int y = (int) (pf.getImageableY() + 4 * dy);
iEnd = HomeInventory.entriesPerPage * (pageIndex + 1);
if (iEnd > HomeInventory.numberEntries)
    iEnd = HomeInventory.numberEntries;
for (i = 0 + HomeInventory.entriesPerPage * pageIndex; i < iEnd; i++)
{
    // dividing line
    Line2D.Double dividingLine = new
Line2D.Double(pf.getImageableX(), y, pf.getImageableX() + pf.getImageableWidth(), y);
    g2D.draw(dividingLine);
    y += dy;
    g2D.setFont(new Font("Arial", Font.BOLD, 12));
g2D.drawString(HomeInventory.myInventory[i].description, (int) pf.getImageableX(), y);
    y += dy;
    g2D.setFont(new Font("Arial", Font.PLAIN, 12));
    g2D.drawString("Location: " + HomeInventory.myInventory[i].location, (int)
(pf.getImageableX() + 25), y);
    y += dy;
    if (HomeInventory.myInventory[i].marked)
        g2D.drawString("Item is marked with identifying information.", (int)
(pf.getImageableX() + 25), y);
    else
        g2D.drawString("Item is NOT marked with identifying information.", (int)
(pf.getImageableX() + 25), y);
    y += dy;
    g2D.drawString("Serial Number: " +
HomeInventory.myInventory[i].serialNumber, (int) (pf.getImageableX() + 25), y);
    y += dy;
    g2D.drawString("Price: $" + HomeInventory.myInventory[i].purchasePrice + ",

```

Purchased on: " + HomeInventory.myInventory[i].purchaseDate, (int) (pf.getImageableX() + 25), y);

y += dy;

g2D.drawString("Purchased at: " +

HomeInventory.myInventory[i].purchaseLocation, (int) (pf.getImageableX() + 25), y);

y += dy;

g2D.drawString("Note: " + HomeInventory.myInventory[i].note, (int)

(pf.getImageableX() + 25), y);

y += dy;

try

{

// maintain original width/height ratio

Image inventoryImage = new

ImageIcon(HomeInventory.myInventory[i].photoFile).getImage();

double ratio = (double) (inventoryImage.getWidth(null)) / (double)

inventoryImage.getHeight(null);

g2D.drawImage(inventoryImage, (int) (pf.getImageableX() + 25), y, (int) (100 * ratio), 100, null);

}

catch (Exception ex)

{

// have place to go in case image file doesn't open

}

y += 2 * dy + 100;

}

return PAGE_EXISTS;

}

}

InventoryItem.java:

package homeinventory;

public class InventoryItem

{

public String description;

public String location;

public boolean marked;

```
public String serialNumber;  
public String purchasePrice;  
public String purchaseDate;  
public String purchaseLocation;  
public String note;  
public String photoFile;
```

```
}
```

Suven Internship