Problem 1: Image Factory (Factory Method and Singleton)

An application needs to create icons and images for display on the UI. Creating images can be complicated and you may want to *change* the way images are stored. JavaFX has an Image and ImageView class for representing images. An ImageView is a resizable Image, and it is used for icons on Buttons and other Controls.

Here's a simple (but not very good) way to use images:

```
Image image = new
Image( getClass().getResourceAsStream("like.png") );
ImageView iview = new ImageView( image );
// constructor: Button(String text, Node graphic )
// ImageView is a subclass of Node.
// Set the text to null to show only an image
Button likeButton = new Button( null, iview );
```

You can also add an image to a control after creation:

```
button.setGraphic( imageView );
```

1.1 *Encapsulate* the method of creating images to avoid duplicate code and accommodate *change.* Define a class for creating images using the *Factory Method* pattern. Define an ImageFactory with a getImage(String) method. We only need one ImageFactory, and want to allow *subclasses (to enable specializing the image factory)* so apply the Singleton Pattern with protected constructor.

Example code for using ImageFactory is shown here:

```
// Single instance of factory
ImageFactory factory = ImageFactory.getInstance( );
// Get an Image. We just give it the image name and the
// factory will decide how to create it.
// "dog" may be in file "dog.jpg", "dog.png", "dog.gif", etc.
ImageView dogIcon = factory.getImage( "dog" );
Label dog = new Label("Dog", dog);
```

1.1 Write an **`ImageFactory`** class that uses the *Singleton Pattern*. It should have a *protected* constructor and a static getInstance() method.

1.2 Write a getImage(String name) method to return an ImageView having the given name.

1.3 getImage(name) must find the image file for name using the CLASSPATH (so images can be location-independent). getImage should append common file extensions to the String name.

Example: `getImage("cat")` will try to create an icon from "`cat.png`", "`cat.jpeg`", and "`cat.gif`" using the CLASSPATH.

Hint: Use Class getClassLoader().getResource() to create a URL for the image file. If the URL is null it means no such file. See Javadoc for ImageView on how to create icons.

Problem 2: Caching ImageFactory

The same Image may be used many times in the application. We don't want to create extra objects, so the ImageFactory should remember which images it has already created. If the application asks for the same image again, the factory returns a reference to the images it already created. This is called a *cache*, just like a *browser cache* that stores recently requested objects from the Internet.

2.1 Create a *subclass* of `ImageFactory` named `CachingImageFactory`. The subclass *overrides* the `getImage(String)` method to add ability to store images in a Map and get them

from the map. The *first time* that `getImage("dog")` is called, it creates a returns the Image and adds it to a *Map* of known images. After that, it uses Images from the Map.

2.2 `getImage` should **not duplicate the code** of the superclass `getImage` method. When it needs to create a new image, invoke the superclass method.

2.3 Modify `getInstance()` in `ImageFactory` to create an instance of CachingImageFactory. But **don't modify the method signature**! The application should *not know about the subclass*.

Hint: Use a Map<URL, Image> or Map<String, Image> to store reference to Images that you create.

Note for Applications: ImageView is mutable but Image is immutable. So it is safe to reuse the same Image, but (depending on the application) it might not be safe to return the same ImageView. For the exam, you can just return the same ImageView, but for an application you would *cache* the Image object and create a new ImageView each time getImage is called.

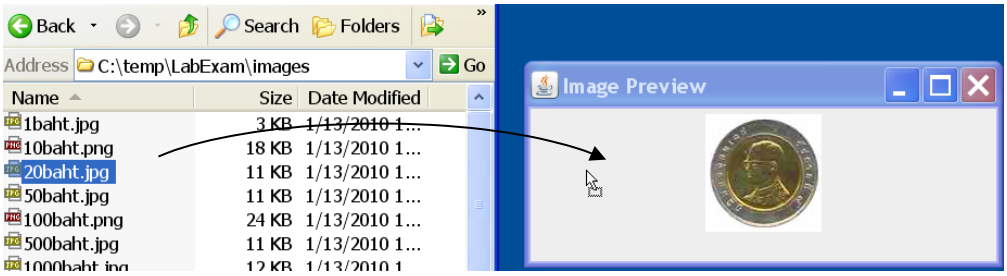
Example use:

```
// Singleton instance of factory
ImageFactory factory = ImageFactory.getInstance( );
Icon dog = factory.getImage( "dog" );
Icon cat = factory.getImage ( "cat" );
Icon dog2 = factory.getImage ( "dog" );
// Test that the icon is the same (using a cache)
if (dog ==  dog2) System.out.println("Correct! Icons are same");
else System.out.println("Wrong. Should create only one dog image");
if (dog == cat) System.out.println("Are you crazy? dog != cat");
```

Problem 3: Graphical Image Viewer with Drag & Drop (To Be Updated for JavaFX: This problem uses Swing)

Download: Download `filedrop-1.1.zip` from the class week15 folder. Unzip it and add **`filedrop.jar`** to your project.

Create a GUI window that shows images of graphics files (jpg, gif, png). The Window shows images in a row. The user can *drag and drop* image files into the window.



To enable drag & drop, use the FileDrop class and write a *Listener* class that implements FileDrop.Listener.

1. Name the image viewer class ImageViewer and include a main method to launch the app using the standard pattern for GUI apps we studied in labs.

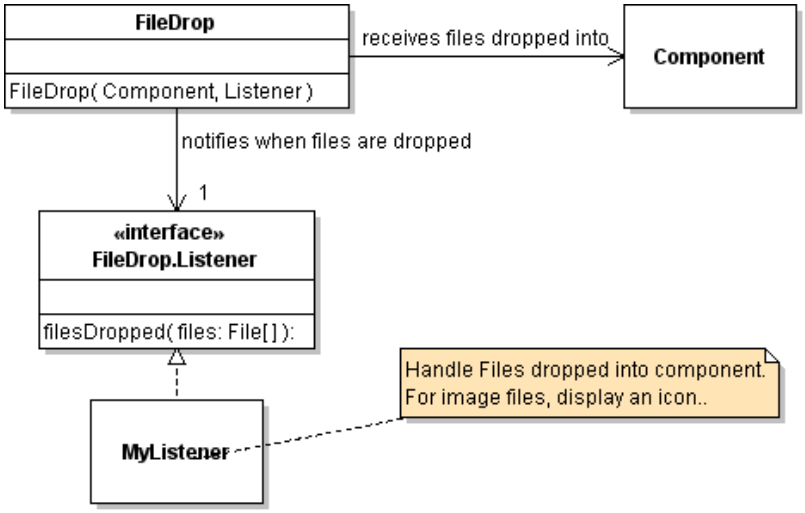
2. Create a window containing a JPanel inside a JScrollPane. Enable *horizontal scroll bars* in the JScrollPane as needed (scrollbar appears only when needed).

3. Use FileDrop to enable dragging & dropping of images into the window. Your application should allow dropping multiple files at the same time, and don't throw exception or other error if the file cannot be displayed as an Icon!

4. Write a class that implements FileDrop.Listener to receive notification of files dropped into the window. This listener class should use ImageFactory to get an Icon for each file dropped

If ImageFactory can create an icon then add it to the panel; otherwise, do nothing. Use JLabels to show the Icons in the panel.

Note: JScrollPane is a *decorator* (wrapper) that adds scroll bars to a JPanel. JPanel is the container for images. You should add icons (as JLabels) to the JPanel, *not* the JScrollPane.



Using FileDrop

FileDrop is an *adapter* for Java's Drag and Drop. To use FileDrop you need to do 3 things.

1. Add filedrop.jar to your project.

2. Write a class that implements FileDrop.Listener. See the Javadoc below.

3. Create a FileDrop object attached to the Component or JComponent you want to receive drag & drop. You do this in the constructor (see Javadoc below).

You can use *any Component* to receive file drop, not just JPanel.

Handling File names in ImageFactory

FileDrop.Listener receives an array of File objects.

For the ImageFactory, you need to convert those File objects to something that ImageFactory can use to create (and map) images.

You could get the file's path using `file.getPath()`, `file.getCanonicalPath()`, or `file.getAbsolutePath()`. But the String path probably **won't work** with ImageFactory.getIcon( ).

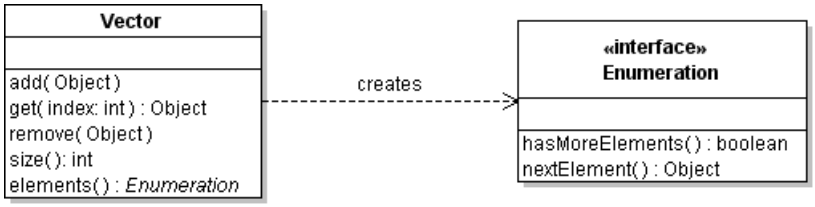
Solution: add a new method to ImageFactory and CachingImageFactory:

`Icon getIcon(URL imageurl)` - create an icon from URL and return it

To avoid duplicate code, the getIcon(String) method should convert the String to a URL using getResource(), and the invoke getIcon(URL). For File objects, File has to toURI() method.

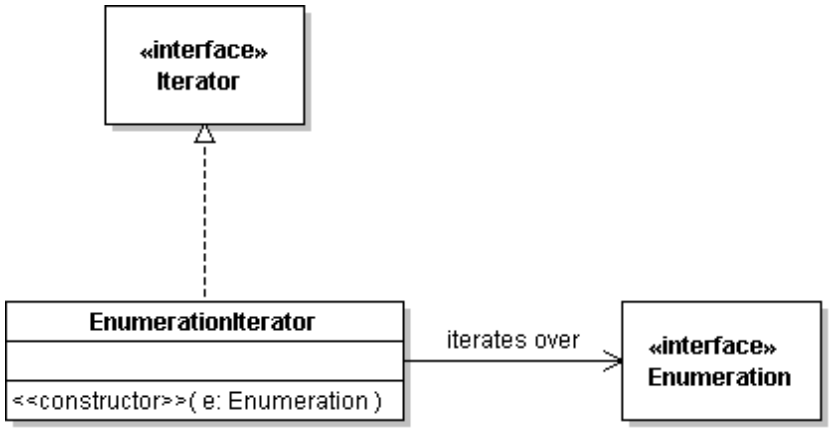
Problem 4: Enumeration Adapter

Many older Java classes (such as Vector) use an *Enumeration* for collections of objects. However, newer Java classes use *Iterator* instead. We would like to be able to convert an *Enumeration* to work like an *Iterator*.




```
Vector<String> fruit = new Vector<String>( );
fruit.add( "Apple" );
fruit.add( "Banana" );
fruit.add( "Orange" );
// Vector returns an Enumeration, not an Iterator
Enumeration<String> e = fruit.elements( ); // create Enumeration
while ( e.hasMoreElements( ) ) {
   System.out.println(  e.nextElement( ) );
}
```

Write an *adapter* for Enumeration that provides an *Iterator* interface for an Enumeration object.



Test your **`EnumerationIterator`** class using the fruit example above.

```
Enumeration e = fruit.elements();
Iterator iter = new EnumerationIterator( e );
while( iter.hasNext() ) {
     System.out.println( iter.next() ); // should print the fruits
}
e.hasMoreElements()  // should return false
```

Problem 5: Calculator with Commands

Source Code: Download CalculatorWithoutCommands.zip from week14 folder. Unzip it and add the files to your project for this problem.

Description: The source code contains a simple calculator (it only has + and - operators) and a console interface. A calculator stores 2 numbers and a pending operator, like + or *. The two numbers are stored in register1 and register2 of the calculator.

Here is an example of input and what the calculator would store:

Input	Register1	Register2	Operator
12			**Noop**
+	**12**		**+**
20	**12**		**+**
=	**32**	**20**	**+**

When you press = (or ENTER on console) the calculator stores the current value in register2 and performs the pending Operator. It stores the result of the operation in register1. (This enables you to repeat the previous operation by pressing "=" again.)

To Do: Make a copy of CalculatorWithoutCommands named Calculator. Put your solution in Calculator.

5.1 The sample code stores the operator as a character. The **performOperator** method of Calculator looks like this:

```
/** perform operation and return the result */
protected double performOperator() {
     double result = 0;
          switch (operator) {
          case '+':
               result = register1 + register2;
               break;
          case '-':
               result = register1 - register2;
               break;
          case '=':  // NOOP. Just return first register value.
               result = register1;
               break;
          default:
               throw new IllegalArgumentException(
                                   "Unknown operator " + operator);
          }
        return result;
    }
```

Apply the Command Pattern to this code and eliminate the switch statement.

(a) Define an interface for Operators. The Operator interface should have one method: apply(x,y) that performs the operation on arguments x, y and returns the result.

(b) Modify the Calculator class to use Operator objects instead of char.

(c) Modify ConsoleUI to send Operator objects to Calculator instead of char. (setOperator)

Test your code.

5.2 OperatorFactory: write an OperatorFactory class that returns operators. It should have 2 methods:

static OperatorFactory getInstance() - create an instance of OperatorFactory and return it.

Operator getOperator(char c) - return an Operator for char c (+, -). Return null if c is not the name of an operator.

We only need *only instance* of each Operator, so the factory should always return the same instance of each operator. That is, `getOperator('+')` always returns the same `AddOperator` object.

5.3 Modify `ConsoleUI` to use `OperatorFactory`. Console UI should not depend on any specific operators. It should not test for "+" or "-" in the input. This is the *Open-Closed Principle.* We want the ConsoleUI to be *extensible.*

5.4 Define new operators without modifying ConsoleUI:

a) Define * and / operators. You should only need to modify the OperatorFactory.

b) Define a ^ operator for powers: 8 ^ 2 = 64

If you need to modify ConsoleUI to use the new operators, then your solution is incorrect.

5.5 Extra Credit: Use Lambda Expressions

The operators are very simple and the factory creates only *one instance* of each operator. Use *lambda expressions* to define the operators inside OperatorFactory. The OperatorFactory can create a single object (say) addOperator, subtractOperator as *anonymous classes* that implements Operator.

```
final Operator add =  /* write a lambda for addition */ ;
```

Problem 6: Coin Machine Logger (Decorator Pattern)

We would like to create a *log* (record) of every time that someone inserts or withdraws money using the Coin Machine. The log should include the time and activity, such as this:

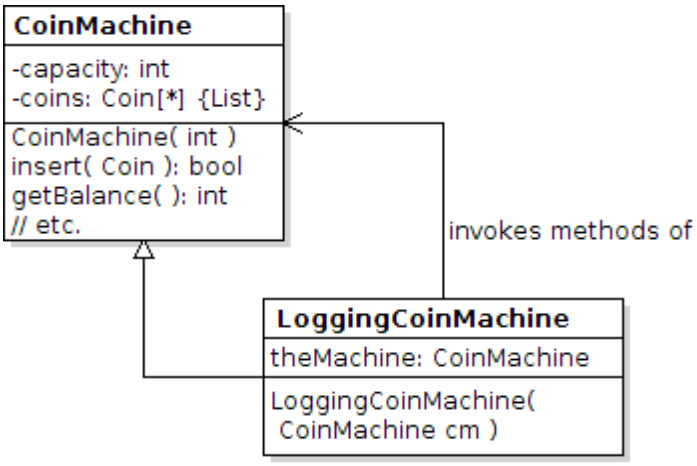
```
17:05:00 insert 5-Baht
17:05:12 insert 20-Baht
17:06:18 withdraw 100-Baht failed
17:08:20 withdraw 20-Baht
```

We don't want to modify the Coin Machine, so instead create a *decorator* (also called a *wrapper*) that encapsulates a real Coin Machine. The decorator invokes methods of the real Coin Machine and logs the results.

Write a decoratore named CoinMachineLogger that *look like* a CoinMachine and also *encapsulate* a CoinMachine. The CoinMachineLogger must *override* all public methods and pass them to the real purse (in some cases a decorator provides its own implementation of the methods, such as `UnmodifiableList`).

Wrong: **Don't pass the calls to the superclass object. You'll get no credit for that, since its not the correct behavior.**

The CoinMachineLogger logs results of insert and withdraw to System.err, as in the example above. Other methods are simply passes to the Purse (no logging).



A decorator *wraps* the real object that its decorates, therefore it must override all public methods and pass them to the encapsulated object.

If you don't override *all* the public methods, the superclass methods will be invoked, which refer to the *wrong object*.

Example:

```
CoinMachine cm = new CoinMachine(20); // 20 is the capacity
cm = new CoinMachineLogger( cm );  // add decorator
cm.insert(  new Coin(5) );
cm.insert( new Coin(10) );
System.out.println( cm.getBalance() ); // not logged
cm.isFull();       // returns false
cm.withdraw(1);    // fails, no 1-Baht coins
cm.withdraw(10);   // succeeds
```

Problem 7: State Pattern

The Calculator (Problem 5) is a good application for using state. The way the calculator reacts when you enter an operator depends on its current state; in particular, if there is already another pending operation then perform that operation first.
a) Define the states of the calculator in problem 5.
b) Define State objects for the calculator so the ConsoleUI does not need to use "if" statements to decide what to do when the user enters a number or operator?

Class FileDrop

public class **FileDrop**
extends java.lang.Object

This class makes it easy to drag and drop files from the operating system onto a Java program. Any java.awt.Component can be dropped onto, but only javax.swing.JComponents will indicate the drop event with a changed border.

To use this class, construct a FileDrop.Listener object (to receive notification of files dropped) and a new FileDrop. There are several constructors. The simplest one requires parameters for the target component and a Listener object to receive notification when file(s) are dropped. For example:

```
JComponent component = new JPanel();
new FileDrop( component, filedropListener );
```

You can specify the border that will appear when files are being dragged by calling the constructor with a javax.swing.border.Border. Only JComponents will show any indication with a border.

You can turn on some debugging features by passing a PrintStream object (such as System.out) into the full constructor. A null value will result in no extra debugging information being output.

Constructor

public **FileDrop**(java.awt.Component component, FileDrop.Listener listener)

Constructs a FileDrop with a default light-blue border and, if *c* is a Container, recursively sets all elements contained within as drop targets, though only the top level container will change borders.

Parameters:
component - Component on which files will be dropped.
listener - Listens for files dropped.

Interface FileDrop.Listener

public static interface **FileDrop.Listener**

Implement this inner interface to listen for when files are dropped. For example your class declaration may begin like this:

```
public class MyListener implements FileDrop.Listener {
      public void filesDropped( java.io.File[] files )
      {
          ... // handle files
      }
}
```

Method Detail

void **filesDropped**(java.io.File[] files)

This method is called when files have been successfully dropped.

Parameters:
files - An array of Files that were dropped.