Allowed Materials for Lab Exam

1. You may use an IDE, Java API docs, and Java Tutorial on the local computer or on se.cpe.ku.ac.th.

2. You may NOT use any other written materials, the Internet, online books or slides, any Java source code. Do not communicate with anyone except instructor & exam proctor.

Problem 1: Icon Factory (Factory Method and Singleton)

An application needs to create icons for display on the UI. Creating icons can be complicated and the way of creating icons may change. To allow flexibility in creating icons, use the *Factory Method* pattern. Define an ImageFactory with a getIcon(String) method. We only want one factory, but want to allow *subclasses* so define a static getInstance() method that always returns the same instance of the factory.

```
// Single instance of factory (may return a subclass object)
ImageFactory factory = ImageFactory.getInstance( );
// Get an ImageIcon. 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.
Icon dog = factory.getIcon( "dog" );
Icon cat = factory.getIcon( "cat" );
```

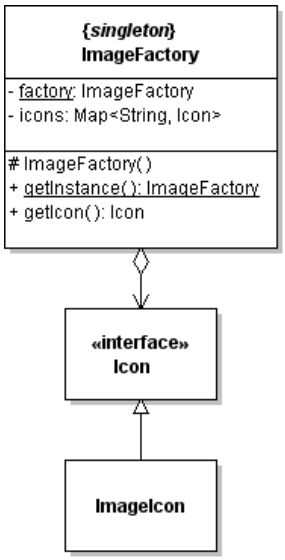
1.1 Write an **`ImageFactory`** class that uses the *Singleton Pattern*. It should have a *protected* constructor and a static getInstance() method.

1.2 Write a getIcon(String name) method to return an icon having the given name. Icon is the interface, ImageIcon is a concrete class.

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

Example: `getIcon("cat")` will try to create an icon from "`cat.jpg`", "`cat.png`", and "`cat.gif`" using the classpath.

Hint: Use class.getResource() to create a URL for the file, use the URL to create ImageIcon. If the URL is null it means no such object.



Problem 2: Caching ImageFactory

The same Icon may be used many times in the application. We don't want to create extra objects, so the ImageFactory should remember which icons it has already created. If the application asks for the same icon again, the factory returns a reference to the icon it already created.

2.1 Create a *subclass* of `ImageFactory` named `CachingImageFactory`. The subclass *overrides* the `getIcon(String)` method to add ability to store icons in a Map and get them from the map. `getIcon("dog")` should only create the icon the *first time* it is requested. After that, return the same dog icon.

2.2 `getIcon` should not duplicate the function of the superclass `getIcon` method. When it needs to create a new icon, just 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 need to know about the subclass.

Hint: Use a Map<URL, Icon> or Map<String, Icon> to store reference to Icons that you create. Using URL as key will work better because it is unique.

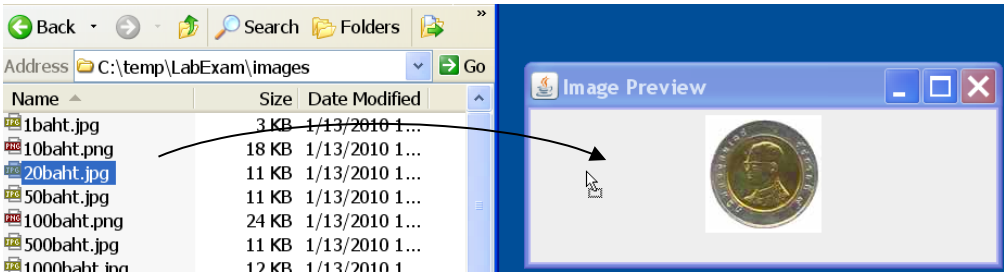
When an Icon is created the first time, the Factory saves it in a Map. The next time the application asks for an Icon that has already been created, the Factory just returns the value from the map.

```
// Singleton instance of factory
ImageFactory factory = ImageFactory.getInstance( );
Icon dog = factory.getIcon( "dog" );
Icon cat = factory.getIcon( "cat" );
Icon dog2 = factory.getIcon( "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 only create one dog icon");
```

Problem 3: Graphical Image Viewer with Drag & Drop

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.

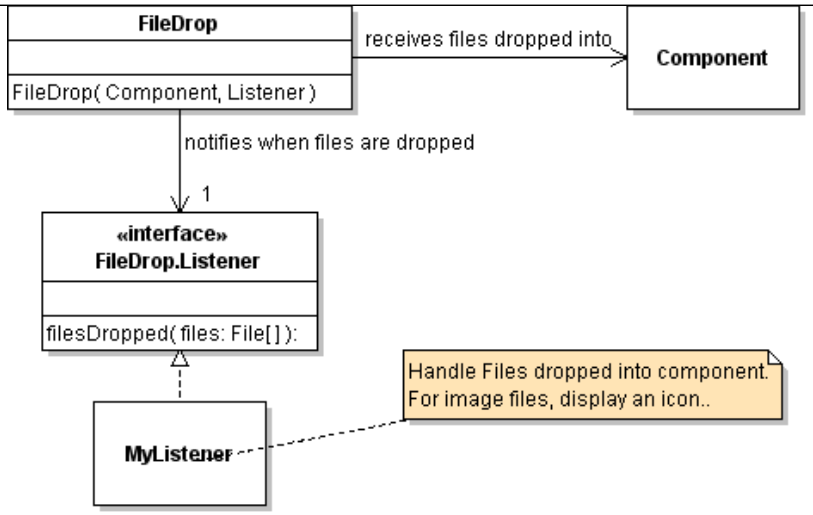
2. Create a window containing a JPanel inside a JScrollPane. Enable *horizontal scroll bars* in the JScrollPane as needed (scrollbar appears only when needed).

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.

The simplest way is to get the name of a File is `file.getPath()`, `file.getCanonicalPath()`, or `file.getAbsolutePath()`. But the String name probably **won't work** with ImageFactory.getIcon( ).

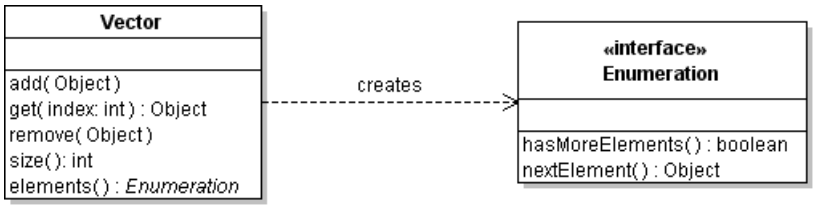
Fix: add another method to ImageFactory and CachingImageFactory:

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

The getIcon(String) method should convert the String to a URL using getResource(), and the invoke getIcon(URL). This avoids duplicate code.

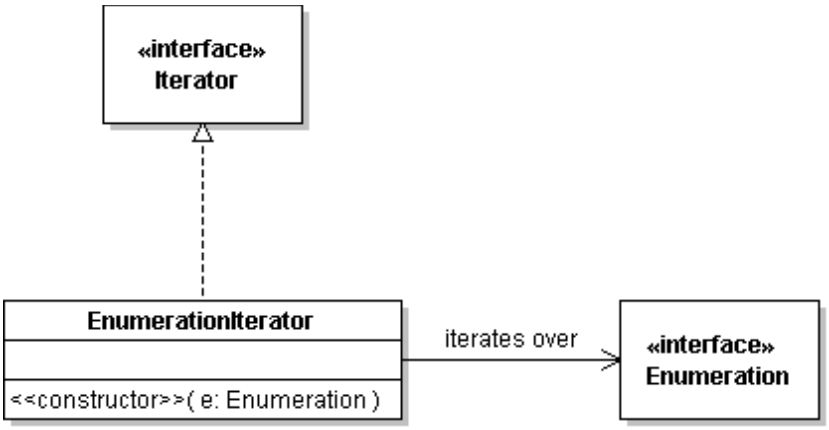
Problem 4: Enumeration Adapter

Many older Java classes (such as Vector) use an *Enumeration* for collections of objects. However, newer Java classes use *Iterator* instead.




```
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 interface should have two methods: perform(x,y) and toString() that returns "+" for addition, "-" for subtraction, etc.

(b) Modify the Calculator class to use Operator 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:

Define MultiplyOperator and DivideOperator. You should only need to modify the OperatorFactory.

Can you 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 Anonymous Classes.

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

```
final Operator addOperator =  new Operator( ) { /* write an anonymous class */ }
```

Problem 6: Purse Logger (Decorator Pattern)

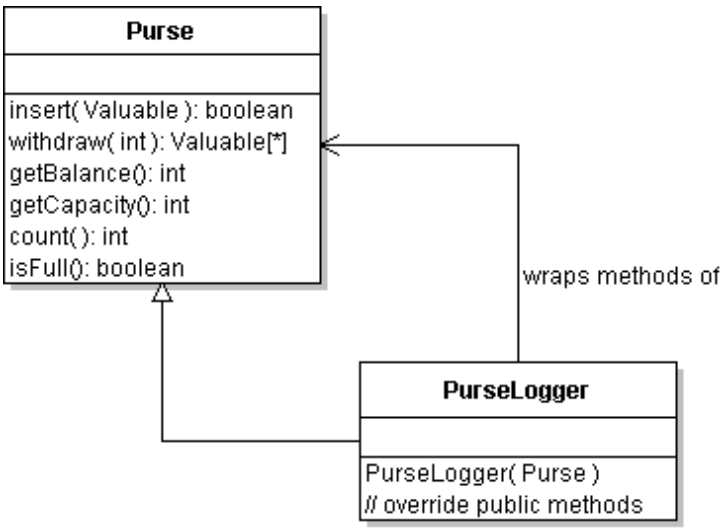
We would like to create a *log* (record) of every time that someone inserts or withdraws money from a Purse object. Other methods like getBalance() do not need to be logged. The log should include the time and activity, such as this:

```
17:05:00 insert 5-Baht coin
17:05:12 insert 20-Baht bank note
17:06:18 insert 1-Baht coin failed
17:08:20 withdraw 20
```

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

Write a decoratore named PurseLogger that *look like* a Purse and also *encapsulate* a Purse. The PurseLogger 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`).

The PurseLogger 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:

```
Purse purse = new Purse(10);
purse =  new PurseLogger( purse );  // add decorator
purse.insert(  new Coin(5) );
purse.insert( new BankNote(20) );
System.out.println( purse.getBalance() ); // not logged
purse.isFull();
purse.withdraw(1);
purse.withdraw(20);
System.out.println( purse.getBalance() );
```

Problem 7: State Pattern

There will probably be an exam problem involving the state pattern using Objects.
The Calculator (Problem 5) is a good example. Can you write 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?

Problem 8: Immutable Set (Decorator)

Write a wrapper for a Set that makes the set immutable. This is useful for a method that must return a Set attribute, but we don't want the application to be able to modify the Set.

```
Set<String> set = new HashSet<String>();
set.add("apple");
set.add("orange");
size.size()
2
// Create an immutable view of the set.
// It encapsulates the underlying set
ImmutableSet<String> iset = new ImmutableSet<String>( set );
iset.contains("apple")
true
iset.size()
2
iset.remove("apple")
UnsupportedMethodException: can't remove
// remove from original set:
set.remove("apple")
true
// did the iset change, too?
iset.contains("apple")
false
iset.size()
1
```

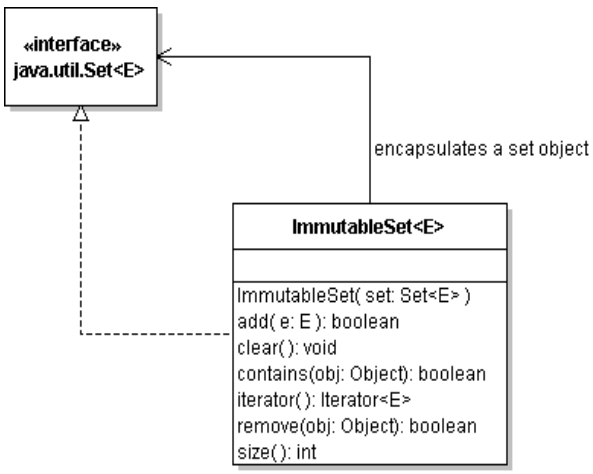
1. Write an `ImmutableSet` that provides the methods shown in the UML diagram. In reality, you would need to override *all* the Set methods, but for this exam, just override the methods shown (to save time).

2. Any methods that change the Set should throw `UnsupportedMethodException`.

3. Other methods should *delegate* to the encapsulated Set. Don't invoke super.method !

4. `ImmutableSet` must have a *type parameter*.

5. You may want to extend `AbstractSet` (for convenience) but are not required to.



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.