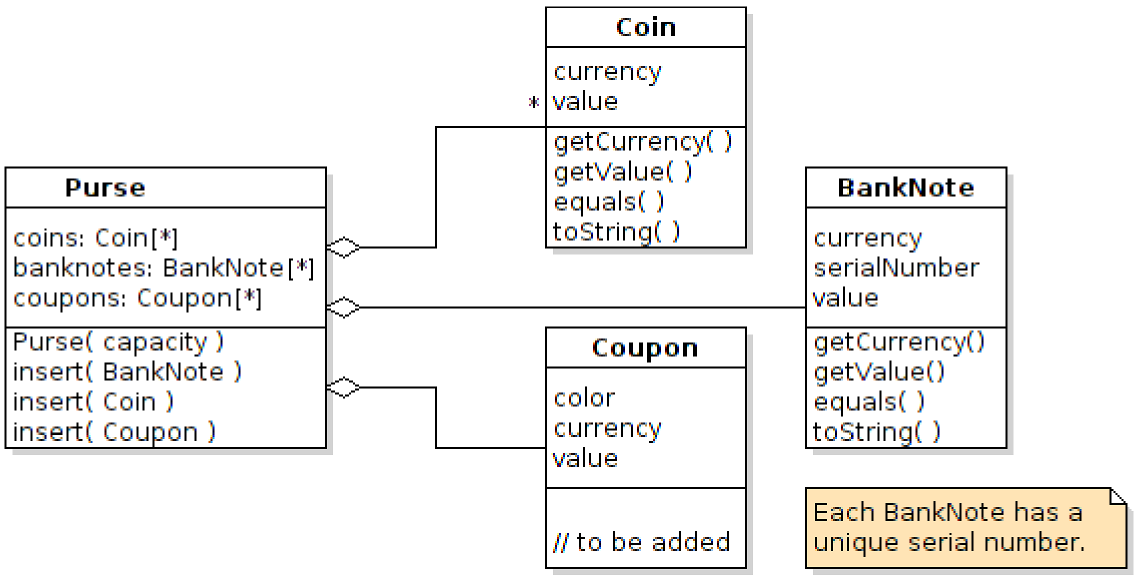
Objectives	Enable the Purse to handle different classes of money. Design an interface to enable this. Modify classes to depend on the interface rather than Coin. Modify CoinUtil.sumByCurrency to eliminate *side effects* (sorting the list).
What to submit	Before starting work for this lab, create a TAG to "bookmark" the completed work for Lab 3 coinpurse. Commit the revised code to the coinpurse project on Github, using the same project as last week.

New Requirements

We want the Purse to be able to store Banknotes and other kinds of monetary objects, such as Coupons for KU Fair.

A Bad Design

A simple solution (but more complex coding) would be to add a separate insert method for different kinds of money.



This design might work, but it is complicated and would have lots of *duplicate logic*.

Exercise 1: Identify Required Behavior of Money

To simplify the design, we need to enable the Purse can treat all kinds of money the <u>same way</u>.
We need to "abstract" the essential characteristics of money and define an interface for that behavior.
Examine the code for Purse and CoinUtil. What Coin methods do they invoke? Then answer this:

Question: <u>What</u> are the required characteristics of money? What does Purse or CoinUtil need to know about Coin (or other money) in order to perform its methods?

Answer: ____

Exercise 2: Define a `Valuable` Interface

The Purse doesn't care *how* a Coin or Banknote determines its value. The Purse only needs to know a ***how* to *<u>ask</u>*** for its value. The CoinUtil also needs to also *ask* the objects for their currency.

2.1 Create an interface named `Valuable` in the `coinpurse` package.

```
package coinpurse;
// TODO write good Javadoc. An interface is a specification.
// A specification needs documentation so that others can use it.
/**
 * An interface for objects having a monetary value and currency.
 */
public interface Valuable {
     /**
      * Get the monetary value of this object, in its own currency.
      * @return the value of this object
      */
     public double getValue( );
//TODO write getCurrency
}
```

The *real* money classes (Coin, BankNote, and others) also need some other methods:

toString()	so that the user interface can display a description of what it withdraws
equals( )	so we can find objects in a List. equals is called by `list.remove(Object)`.

Exercise 3: Define **`Coin`** and **`BankNote`** that implement Valuable

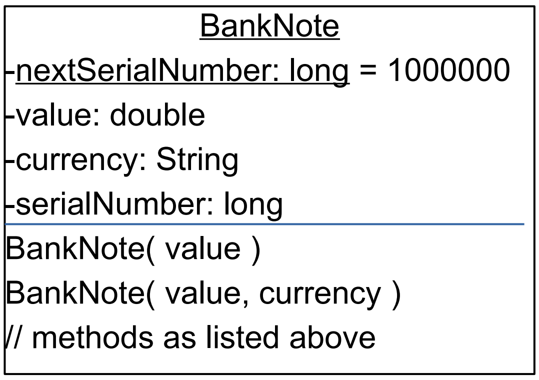
3.1 Modify Coin so that it implements Valuable. The methods are the same as in previous lab.

3.2 Write a BankNote class. The BankNote constructor should assign each Banknote a unique serial number, starting from 1,000,000.

`getValue( )`	return the value of this BankNote.
`getCurrency( )`	return the currency
`getSerial( )`	return the serial number
`equals( Object obj )`	return true if **`obj`** is a BankNote and has the same currency and value
`toString()`	returns "**`xxx-currency note [`*`serialnum`*`]`**"

3.3 Make the serial numbers unique.

In the BankNote class defines a <u>static</u> variable named nextSerialNumber for assigning unique serial numbers. (This is *not* a great design. It would be better to have a factory class that creates Banknotes and assigns serial numbers to them. Just like the national Treasury Office does.)


BankNote
-nextSerialNumber: long = 1000000
-value: double
-currency: String
-serialNumber: long
BankNote(value)
BankNote(value, currency)
// methods as listed above

Exercise 4: Modify Purse to use Polymorphism

4.1 Modify the Purse class so that is will accept anything that is Valuable. Change "Coin" to "Valuable" everywhere in the Purse code. When you are done, the word "coin" should not appear anywhere in the Purse, not even in comments. One exception: OK to mention "Coin" in the class Javadoc comment (but not required).

Note: You can declare a List or array using an interface type. For example:

```
  List<Valuable> money;
  Valuable[] array = new Valuable[20];
```

4.2 In order to sort money, the Purse needs to sort Coin, Banknote, and other valuable. Define a Comparator<Valuable> that will order money the way that is useful for your Purse's withdraw method.

It is also possible to avoid sorting! Whenever you insert an object into the Purse, add it to the list in order of increasing (or decreasing) value. Then the list will always be sorted! You won't need to call sort. The List has an **`add`** method like this:

`list.add(int k, Object obj)` - add obj at position k in the list. Other items are pushed down.

Exercise 5: Test Your Code and Modify the ConsoleDialog

5.1 Write code to test your Purse. Write *at least* one test method for insert, withdraw, and getBalance to test that they work correctly with Coin and Banknote. You can write your own test class or modify the PurseTest (JUnit) class.

5.2 Modify the **`depositDialog`** method of the `ConsoleDialog` class. If the user inputs a value of 20 or more, create a Banknote instead of a Coin.

5.3 Update **`withdrawDialog`** to match changes in the Purse's withdraw method.

Exercise 6: Modify CoinUtil

6.1 Update the CoinUtil class so that all methods accept any objects that implement Valuable. When you are done, the word "Coin" should not appear anywhere except the name CoinUtil.

6.2 In the method `sumByCurrency`, **do not sort the list.** In the previous lab we sorted the list to make it easy to sum. But that causes a *side-effect* of the method. It modifies the list in a way that the caller may not expect.

Use a Map to keep track of the sum for each currency. Map from currency (String) to the sum (double). Accessing a map is slower, so if you are clever you can avoid accessing the map too often (only when the currency of the current object differs from the previous object).

Example: use a map to count word occurrences. It reads data from a scanner and counts 1 each time a word is seen. If you run this, press CTRL-D to end the input from console.

```
Map<String, Integer> wordmap = new HashMap<String,Integer>( );
while( scanner.hasNext( ) ) {
     String word = scanner.next( );
     // update the current count.  If word is not found, then get 0.
     int count = wordmap.getOrDefault( word, 0 ) + 1;
     wordmap.put( word, count );
}
// print all the words (keys) and their counts
for(String word : wordmap.keySet() ) {
     int count = wordmap.get( word );
     System.out.printf("%s %n\n", word, count);
}
```

Git History of Commits

To view the history of a repository use any of these commands:

```
> git log
> git log --pretty=oneline
> git log --graph
```

For a nicer display, including branches and tags, use:

```
> git history   (same as git log1)
```

The long hex numbers (e53e01c6e87abfc4ff238) are the commit id numbers. You can abbreviate them to the first 6 characters.

Ref: https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History

Git: How to use **tags**

A Git tag is a name you attach to a git commit. Tag act as a bookmark so you can locate and checkout a particular revision of your code at any future time. Projects use tags to bookmark releases of code, milestones, and bug fixes. There are 2 kinds of tags: *lightweight* (only a tag name) and *annotated* (tag with a name, description, author, and date).

How to Assign a Tag and Push it to the Remote

Create a tag named "lab3" to bookmark your solution to Lab 3

1. Check that you have committed all your work for Lab 3.

2. Create a tag named "lab3"..

```
> git tag -a lab3 -m "Solution to Lab 3 purse assignment"
```

3. Show all the tags in the local repo:

```
> git tag
lab3
```

4. By default, tags are stored in the local repository and not "pushed" to the remote repository. To push the tag(s) to the remote repository use:

```
> git push --tags
```

5. If you view your repo on Github, in the combobox that shows branches you can also select a tag to display.

Ref: https://git-scm.com/book/en/v2/Git-Basics-Tagging