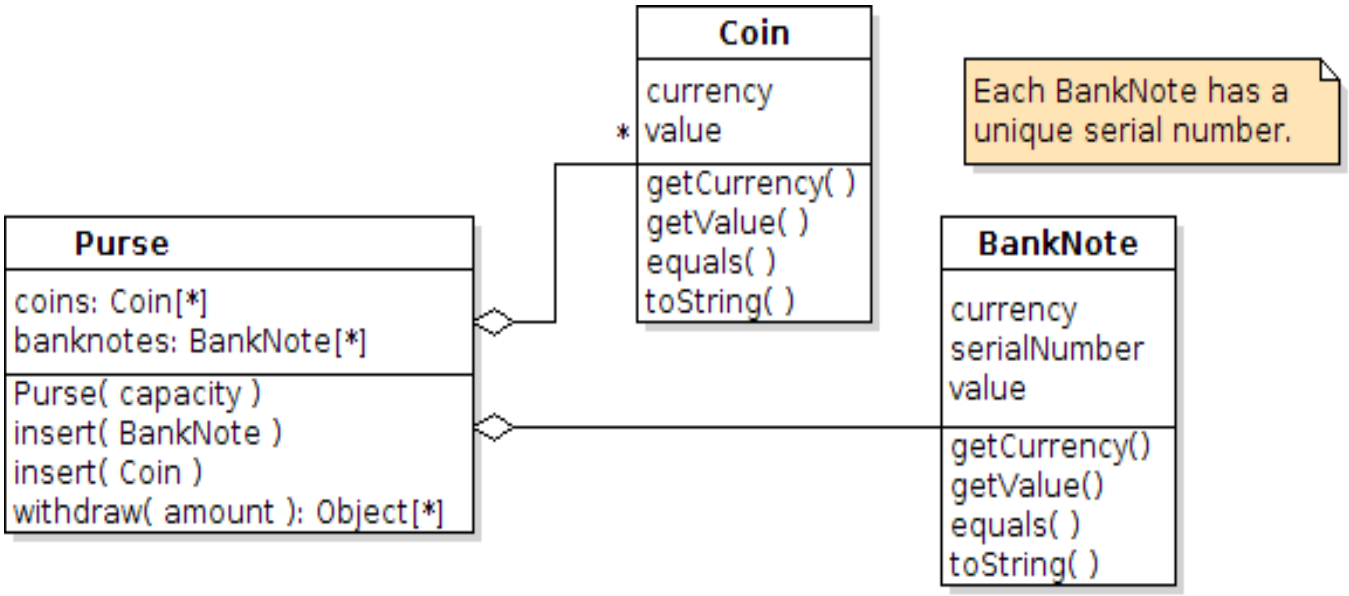
Objectives	1. Enable the Purse to handle different classes of money by creating an *interface* for the required behavior, and modify Purse to depend only on this interface. 2. Write a *Comparator* for money based on the money interface, so the Purse does not depend on the "compareTo" method of different concrete classes. 3. Write another class (BankNote) to demonstrate that polymorphism is working and Purse doesn't depend on the Coin class. 4. Modify MoneyUtil.sumByCurrency to eliminate *side effects* (sorting the list).
What to submit	Before starting this lab, in your coinpurse repository create a Git **Tag** named **`LAB2`** to bookmark the completed code for the Lab 2 Purse. See the end of this lab sheet for how to create tags and "push" tags to Github. Complete and test this lab, then commit the revised code to the coinpurse project on Github, using the same project as last week. c

New Requirements

We want the Purse to be able to store Banknotes and other kinds of monetary objects,.

A Bad Design

A simple solution is to add a separate insert method for BankNote. The withdraw() method must also be modified because the Purse might withdraw either Coins or BankNotes.



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

Better Design: Polymorphism

A better design is to make Coin and BankNote "look the same" to the Purse, so it can handle both using polymorphism. To enable polymorphism you can use either *inheritance* or an *interface*. In this lab, we'll use an interface since it is more flexible.

The first step to enable polymorphism is to identify what *behavior* the application depends on that different kinds of objects (the polymorphic types) must supply. What *behavior* of Coins does the Purse use? Purse uses the Coin's getValue() and getCurrency() methods. So the *interface* for items in the Purse should specify a getValue() and getCurrency() method.

Purse also needs to sort the money. In Lab 2 you made the coins sortable by implementing the *Comparable* interface in Coin -- writing a compareTo method in Coin. In this version of the Purse, you'll separate the "ordering" part from the Coin class by writing a separate *Comparator* (Problem 3.2).

Problem 1: Define a `Valuable` Interface

The Purse needs to know a ***how* to *ask*** different kinds of money for their value and currency. It doesn't care how they determine it -- just how to ask for it. Since an *interface* specifies behavior (without implementing it), *interface* is ideal for this.

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

```
package coinpurse;
// TODO write good Javadoc. An interface is a specification,
// so it needs good documentation!  An interface without documentation
// is USELESS. Write all the tags (@param, @return) for methods.
/**
 * 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
      */
     //TODO write getValue() and getCurrency()
}
```

Real money classes (Coin, BankNote, etc.) should also have these 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)`.

toString() and equals() are already specified in the Object class, so we don't need to add them to the interface.

Problem 2: Declare that `Coin` and `BankNote` implement Valuable

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

2.2 Write a BankNote class. The BankNote constructor has 2 parameters (value and currency). The constructor should also 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 (**`long`**)
`equals( Object obj )`	return true if **`obj`** is a BankNote and has the same currency and value
`toString()`	returns "**`xxx-`*`Currency`*` note [`*`serialnum`*`]`**"

2.3 Make serial numbers unique -- each BankNote has a different serial number.

Hint: define a private static variable that contains the next serial number, e.g. named nextSerialNumber. 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. You'll do that later.

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

Problem 3: Modify Purse to use Valuable instead of Coin

3.1 Modify the Purse class so that is will accept anything that implements Valuable. When you are done, the word "Coin" or "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;                  // list of Valuable
Valuable[] array = new Valuable[20];   // array of Valuable
```

3.2 To sort items in the purse, you need a way to "order" all kinds of money. The Collections class has a sort method that accepts a *Comparator* as second parameter:

```
Collections.sort( List<E> list, Comparator<E> comparator )
```

E is the type parameter. The value of E is the type of object that the Comparator will compare. In this application, "E" is Valuable.

```
package coinpurse;

public class ValueComparator implements Comparator<Valuable> {
    /**
     * Compare two objects that implement Valuable.
     * First compare them by currency, so that "Baht" < "Dollar".
     * If both objects have the same currency, order them by value.
     */
    public int compare(Valuable a, Valuable b) {
        // your code for compare
    }
}
```

In the Purse class you can create a ValueComparator object anyplace you like. The Comparator has no attributes and never changes, so the Purse *really* only needs 1 instance of it. So define it as a private attribute (instead of a local var). For example, if you create the comparator *inside* withdraw method:

```
public Valuable[] withdraw(double amount) {
    Comparator<Valuable> comp = new ValueComparator();
    . . .
}
```

Then it will create a new Comparator object each time withdraw is invoked. That's a waste of time. If (instead) you define it as an attribute of Purse, then the Comparator object is created only once.

Problem 4: 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.

Problem 5: Modify MoneyUtil

Modify the filterByCurrency method so it does not depend on Coin.

static List<Valuable> filterByCurrency(List<Valuable> money, String currency)	Return a List of the elements from the parameter List (money) with currency that is the same as the currency parameter. If no matches, return an empty list.

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 tag - only a tag name, no commit message or other info

annotated tag - tag with a name, description, author, and date

Annotated tags are more useful and what we will use. To create an annotated tag specify the "-a" option when you create the tag. The command is: `git tag -a tag_name -m "describe the tag"`

How to Assign a Tag and Push it to the Remote

Create a tag named "LAB2" to bookmark your solution to Lab 2 (version 1 of the Purse code).

1. Check that you have committed all your work for Lab 2 and made fixes suggested by the TAs.

2. Create a tag named "LAB2"..

```
> git tag -a LAB2 -m "Solution to Lab 2 purse assignment"
```

3. Show all the tags in the local repository:

```
> git tag
LAB2
```

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

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

5. When you view your repo on Github, in the combo-box that shows *Branches* also has a tab to show *Tags*. You can use this to display any tagged revision of the code!

Assign a Tag to a Previous Commit

If you want to assign a tag to a previous commit (rather than the current HEAD) you can specify the revision number as an extra parameter to "git tag".

1. Find the revision number you want to tag. You can find this using "git history" or by looking at the history of commits on Github. Each commit has a *hashcode* that you use to identify the commit. Github and "git history" show the first 7 digits of the hashcode, such as "02e37b0". This is enough to uniquely identify the commit.

2. Suppose you want to tag the revision with id (hash) 02e37b0. You would type:

```
> git tag -a LAB2 02e37b0 -m "describe this revision"
```

Remove a Tag

If you make a mistake in assigning a tag, or you want to move the tag to a different commit, you can delete it using "git tag -d *tagname*", for example:

```
> git tag -d LAB2
```

Reference to Learn More

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