Objectives	1. Enable the Purse to handle different classes of money by creating an *interface* for money, and modify Purse to depend only on this interface. 2. Write a *Comparator* based on the new interface, for ordering different types of objects by currency and value. 3. Write another class (BankNote) to demonstrate that polymorphism is working and Purse doesn't depend on the Coin class.
What to submit	1. Before starting this lab, create a Git **Tag** named **`LAB5`** to bookmark the completed code for Lab 5. Push the tag to Github. See last page for instructions. 2. Complete and test this lab, then commit the revised code to the same repository as previous coin purse.

New Requirements

We want the Purse to be able to store different kinds of "money" in the purse, including Bank Notes, Vouchers with cash value, or even checks.

Design for Polymorphism

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. Then define either an interface or a base class (often an abstract class) that contains the required behavior.

What *behavior* of Coins does the Purse use? That behavior should be specified in the interface.

Purse also needs to sort the money. In the previous lab, you made the coins sortable by implementing the *Comparable* interface in Coin. In this version we separate the task of "comparing" money into its own class, by writing a separate *Comparator* class (Problem 4). This way, the Purse does not depend on Coin or other Money to define "compareTo" the way that the Purse wants.

Problem 0: Assign a Git Tag to Lab5 Purse

Before starting this lab, in your coinpurse repository create a Git **Tag** named **`LAB5`** to bookmark the completed code for Lab 5. See last page of this assignment for how to add a tag and "push" it to Github.

Problem 1: Define a `Valuable` Interface

The Purse depends on getting the currency and value from money. The Purse doesn't care *how* objects determine their value -- just how to ask for the value and currency. An *interface* is ideal for this.

1.1 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 Javadoc tags (@param, @return) in 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
      */
     double getValue();
```

```
    //TODO  write getCurrency()
}
```

Problem 2: Declare that `Coin` implements Valuable and Define a BankNote

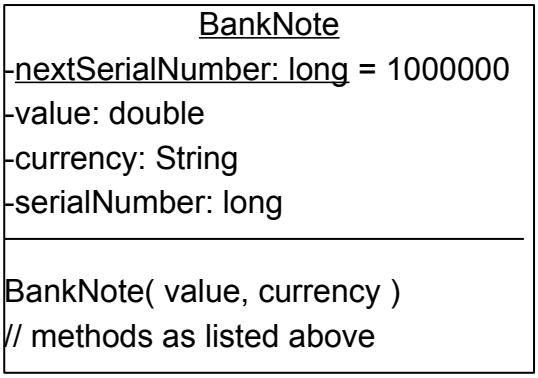
2.1 Modify Coin so that it implements Valuable. The methods are the same as in previous lab.

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. The constructor can use this static value to assign a serial number, then increment it by one. 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.



Problem 3: Modify Purse to use Valuable instead of Coin

Modify the Purse class so that is will accept anything that implements Valuable.

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
```

Problem 4: Write a ValueComparator that implements java.util.Comparator

4.1 To sort items in the purse, you need a way to "compare" all kinds of money (not just Coin). 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. In this application, "E" is Valuable. Collections.sort uses the comparator to decide how to order objects (instead of calling the object's own compareTo method).

Write a Comparator that orders Valuable objects by currency (ignore case) and value, similar to the compareTo method you wrote in Coin:

```
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
    }
}
```

4.2 In Purse, use a ValueComparator to sort the objects in the Purse.

4.3 In the Purse class you need to create a ValueComparator object for withdraw to use. This can be a local variable in withdraw() or a private attribute that you create only once. Since the ValueComparator has no attributes and never changes, its OK to reuse the private attribute.

Problem 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.

Modify **`depositDialog`** and **`makeMoney`** in `ConsoleDialog`:

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

```
/** Make money having the requested value. */
private Valuable makeMoney(double value) {
   if (value >= 20.0) return new BankNote(value, currency);
   else return new Coin(value, currency);
}
```

You can define other kinds of Valuable and "make" them if you like. Consider a Voucher or eCoin for values that are not standard BankNote or Coin values.

5.3 Modify the **`depositDialog`** method of the `ConsoleDialog` class. It has accept any Valuable from makeMoney.

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

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 an annotated tag named "LAB5" to bookmark your solution to Lab 5.

1. Check that you have committed all your work for Lab 5.

2. Create an annotated tag named "LAB5"..

> **`git tag -a LAB5 -m`** `"Solution to Lab 5 purse assignment"`

3. Show all the tags in the local repository:

> **`git tag`**

```
LAB5
```

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", "git log", "gitk" 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 LAB5 02e37b0 -m`** `"Solution to Lab 5 purse"`

Remove a Tag

If you want to move the tag to a different commit, you can delete the old tag using "git tag -d *tagname*", for example:

> **`git tag -d LAB5`**

To Learn More

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