Purpose	1. Use a *superclass* to extract common behavior and eliminate duplicate code. 2. Practice refactoring in Eclipse. 3. Improve purse API for withdraw().
What to Submit	Commit your code to your "coinpurse" project on Github. 1. Before committing your work, create an annotated tag named "`LAB3`" to bookmark your Lab3 coin purse. See Lab 3 worksheet for how to create a tag. Be sure to *push* the tag to Github! (`git push --tags`) 2. Commit your work to the same project.

The Coin and Banknote class from the previous lab have some duplicate code. To remove duplicate code and make it easier to update classes, you'll create a superclass for money objects.

1. Create a Superclass for MoneyTo eliminate duplicate code, create a class named **Money** that implements *Valuable*.

We want the Money class to provide the value, currency, getValue, and getCurrency for subclasses.

1.1 Review your Coin and BankNote classes -- check that the code for currency, value, getValue(), and getCurrency() is the <u>same</u> in both. If not, modify it.

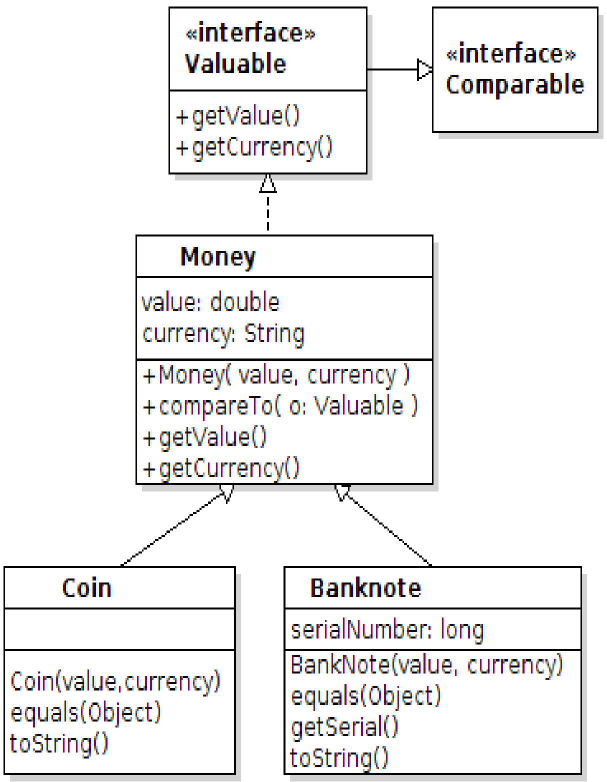
1.2 Create a Money class as shown in the UML.

(a) *Refactoring:* Practice using *Refactoring* in Eclipse or other IDE. Its easy and faster than typing. Instructions at end of this lab sheet.

(b) *Fall-back to Editing:* if you can't get Refactoring to work you can do it manually. But ask TA for help first.

Either (a) or (b), you should move value, currency, and the "get" methods to Money.

1.3 Using either refactoring or manual editing, declare that Coin and Banknote, are subclasses of Money.



1.4 Add a parameterized constructor to Money to set the value and currency. Then, in Coin and BankNote, call `super(value, currency)`. Less duplicate code!

1.5 Coin and BankNote *should not* have a value, currency, getValue, or getCurrency members. They inherit those from the superclass.

1.8 In Coin and BankNote, modify toString() to use getValue() and getCurrency() instead of the value and currency attributes.

1.9 Since Money provides getValue() and getCurrency(), declare that Money implements *Valuable.* The subclasses <u>implicitly</u> implement any interface that a superclass implements.

Therefore, Coin and BankNote should *not* declare "implements Valuable" (its redundant).

2. Refactor equals() - "Pull up" to the Money class

2.1 The equals(Object) method in Coin and BankNote only need getValue() and getCurrency() to perform the equals test. So, move equals() to Money. You may need to modify the code so that equals works correctly for all subclasses. In step 3 of the equals template, *cast* the parameter to be Money or *Valuable* so you can call getValue() and getCurrency() for any subclass that extends Money.

```
if (this.getClass() != obj.getClass() ) return false;
```

```
Money m = (Money) obj;  // cast to Valuable also works
```

Eclipse Refactoring: Select the equals method in Coin or BankNote and choose Refactor -> Pull Up to move the method to the superclass. You can ask Eclipse to "Pull Up" the method from both Coin and BankNote at the same time.

2.2 Verify that the equals() method works correctly in both subclasses (Coin and BankNote).

Note: You should test your code before submitting it. If your equals *doesn't* work for subclasses, the TAs will mark this as incorrect with no chance to correct it.

3. Modify Valuable to *extend* Comparable

In Lab 2 you wrote Coin implements Comparable<Coin>. Since the Purse now accepts other kinds of money, it is more useful for all Valuable objects to be sortable.

3.1 Modify *Valuable* to declare it is also *Comparable<Valuable>*.

3.2 Delete "`implements Comparable`" from Coin and BankNote.

3.3 Write **`compareTo`** in Money so that it works correctly with any *Valuable* objects. Order items by: (a) currency (so items with same currency are grouped together), (b) if two items have the same currency then order by value.

Be careful how you compare numbers! Some currencies may have very small values (0.000001 Bitcoin) and others vary large (100-trillion Zimbabwe dollar note, prior to 2006). The Double class has a static compare method you can use: **`Double.compare(a,b)`**. This is better than (int)Math.signum(a-b) which is harder to understand, and in rare cases may be wrong!

100000000000000
RESERVE BANK OF ZIMBABWE
100000000000000
I promise to pay the bearer on demand
ONE HUNDRED TRILLION DOLLARS
for the Reserve Bank of Zimbabwe
100 000 000 000 000
HARARE 2008
AA3905431

4. Write a withdraw(Valuable amount) for Purse.

The original `withdraw` method did not have a currency. Since money always has a currency, define a withdraw for money with a currency. You need to update the withdraw algorithm to check for matching currency. For backward compatibility, keep the old withdraw(double) and assume the currency is "Baht":

Valuable[ ] withdraw(Valuable amount)	Withdraw the amount, using only items that have the same currency as the parameter (amount). amount must not be null and amount.getValue() > 0.
Valuable[ ] withdraw(double amount)	Withdraw amount using the default currency, which (for now) is "Baht". Don't write duplicate code.

5. Test and Review Your Code

- Coin and Banknote do not have equals or compareTo, they use the methods from Money.
- The Purse does not depend on Coin, BankNote, or Money. It depends only on Valuable. One exception: the withdraw(double) method can create a Money object to pass to withdraw(Valuable).
- Javadoc comments are updated to reflect changes in code. Update parameter names to suggest their meaning. No "insert(Valuable coin)".

Here's an example of **out-of-date** Javadoc and parameter name:

```
/**
 * Deposit coins into the purse.
 * @param coin is the Coin to insert.
 */
public boolean insert(Valuable coin)
```

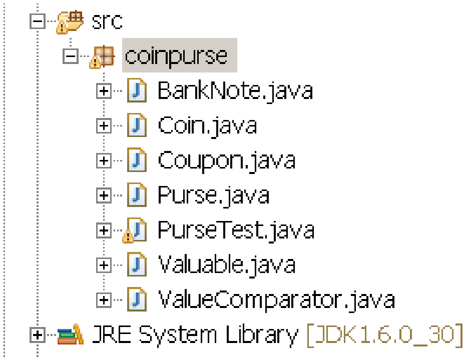
Refactoring in Eclipse

"*Refactoring*" means to restructure source code. Eclipse and other IDE have many refactoring operations to save time & reduce errors. The "Extract Superclass" refactoring creates a superclass and can move methods to the superclass.

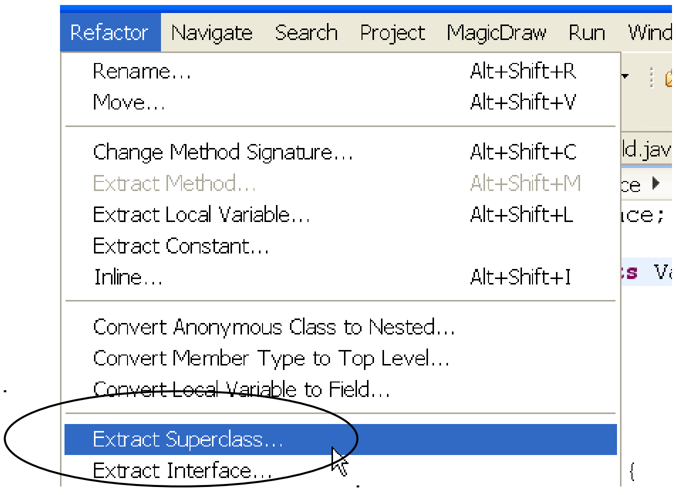
We want to create a superclass for Banknote and Coin.

Eclipse can create a superclass and restructure code for you. Follow these steps:

1. Open the Coin class in the Eclipse editor.



2. From the Refactor menu select Extract Superclass...


Refactor
Navigate
Search
Project
MagicDraw
Run
Wind
Rename... Alt+Shift+R
Move... Alt+Shift+V
Change Method Signature... Alt+Shift+C
Extract Method... Alt+Shift+M
Extract Local Variable... Alt+Shift+L
Extract Constant...
Inline... Alt+Shift+I
Convert Anonymous Class to Nested...
Convert Member Type to Top Level...
Convert Local Variable to Field...
Extract Superclass...
Extract Interface...

3. In the Extract Superclass dialog, enter `Money` as the Superclass name.

3.1 Check the box:

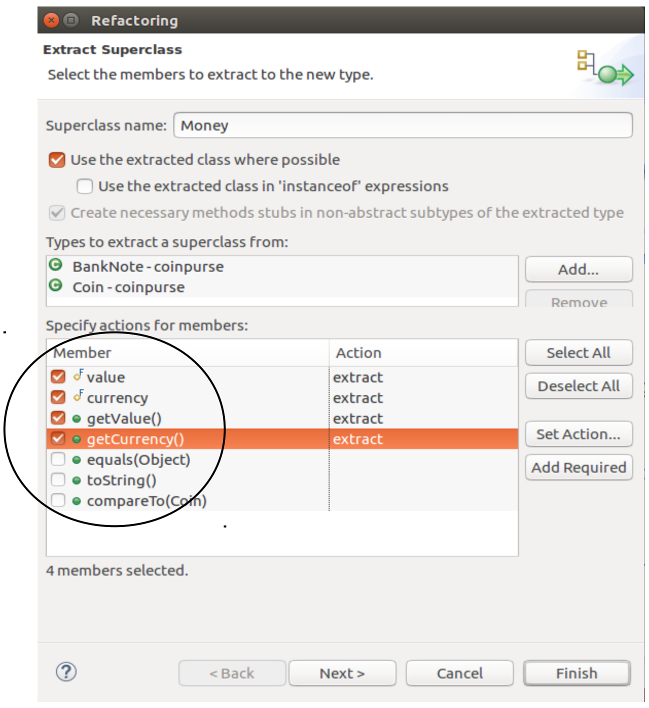
[x] Use the extracted class where possible

4. Click Add... and select BankNote as another class you want to refactor.

5. Select the fields and methods you want to extract. They will be moved to superclass.

Extract: value, currency, getValue(), getCurrency(). They should be the same in both subclasses (Coin and BankNote).

Click the Next> Button.



Refactoring

Extract Superclass

Select the members to extract to the new type.

Superclass name: Money

[✓] Use the extracted class where possible

[ ] Use the extracted class in 'instanceof' expressions

[✓] Create necessary methods stubs in non-abstract subtypes of the extracted type

Types to extract a superclass from:

BankNote - coinpurse
Coin - coinpurse

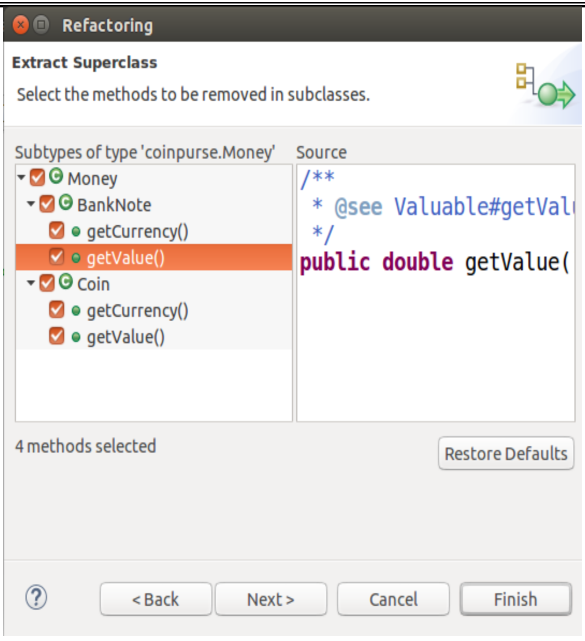
Specify actions for members:

Member	Action
[✓] value	extract
[✓] currency	extract
[✓] getValue()	extract
[✓] getCurrency()	extract
[ ] equals(Object)	
[ ] toString()	
[ ] compareTo(Coin)	

4 members selected.

6. This dialog lets you choose which methods to remove from subclasses.

Remove getCurrency() and getValue() from **both** Coin and BankNote.



Select the methods to be removed in subclasses.

Subtypes of type 'coinpurse.Money'

- ✓ Money
  - ✓ BankNote
    - ✓ getCurrency()
    - ✓ getValue()
  - ✓ Coin
    - ✓ getCurrency()
    - ✓ getValue()

Source

```
/**
 * @see Valuable#getVal
 */
public double getValue(
```

4 methods selected

Click Next> (or Finish).

Notice the warning message about problems. You can fix these problems after refactoring.

7. Fix the code after refactoring.

Because we created Money by refactoring, Eclipse created only a default constructor.

We need a constructor with parameters:

```
public Money(double value, String currency)
```

then the subclass constructors can invoke:

```
public Coin(double value, String currency) {
   super( value, currency);
}
```

How to Extract More Methods to Superclass

You can move methods from a subclasses to a superclass after you have created it. In the Refactor menu choose "Pull Up". This means to move a field or method to a superclass.

You could use "Pull Up" instead of "Extract Superclass" if you create the superclass (Money) first. This sometimes results in less editing later on.

Undo Refactoring

If you make a mistake, you can Undo refactoring using Edit -> Undo, or Refactor -> History.

Note that if you edit the code after refactoring then undoing a refactoring make create errors in code.

A safer way is to commit your code to git *before* refactoring. Then you can revert your working copy to the previous git commit if the refactoring doesn't work.