Purpose	1. Use an *abstract superclass* to implement common behavior and eliminate duplicate code. 2. Practice refactoring in Eclipse.
What to Submit	Commit your code to your "coinpurse" project on Github. 1. Before committing your work, create an annotated tag named "`lab4`" to bookmark your Lab4 coin purse. See Lab4 worksheet for how to create a tag. Be sure to *push* the tag to Github! 2. Commit your work to the same project.

In a previous lab, you wrote an interface for *Valuable* objects to enable polymorphism, and defined several kinds of valuable objects that can be put in a Purse.

This makes the Purse a lot more flexible, but there is some duplicate code.

For example, getValue() and getCurrency() are the same in most or all the classes implementing Valuable. equals() is nearly the same.

1. Create an Abstract Superclass for Money

To eliminate duplicate code, create an Abstract class named **`AbstractValuable`**. This will be the parent class for Coin and Banknote (and may be used by other kinds of money).

Note: Eclipse can do 1.1 - 1.3 by using *Refactoring* . *See below for how.*

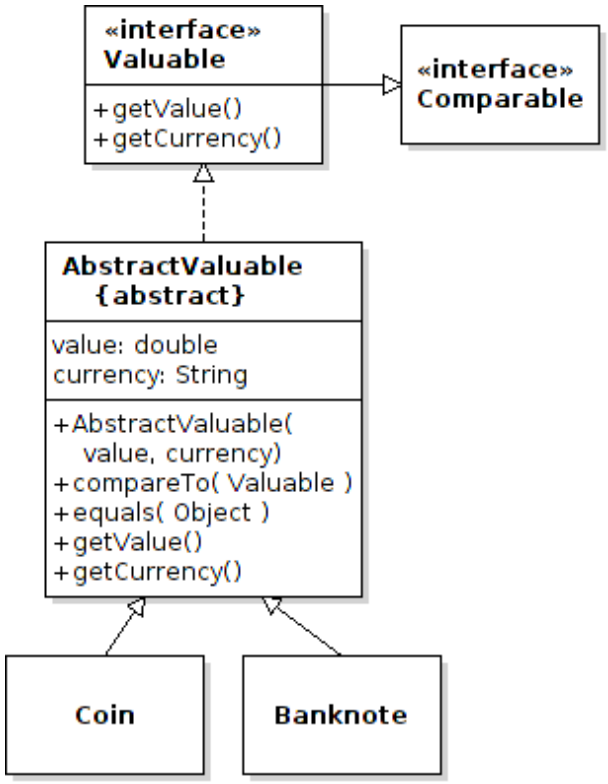
1.1 Declare the AbstractValuable class to be **`abstract`**.

1.2 Declare that Coin and Banknote, are subclasses of AbstractValuable.

1.3 Move methods that are the same in Coin and Banknote to AbstractValuable. If you are using Eclipse, use *Refactoring* to extract the methods.

Verify that **`equals( )`** in AbstractValuable works correctly for each subclass. To verify that two objects belong to the same class use:

```
if (this.getClass() == obj.getClass() ) ...
```



What is a good name?

A class should have a descriptive name. You can name this class AbstractMoney if your prefer.

Note: in the `equals` and `compareTo` of AbstractValuable, you should use `getValue()` to get the value of objects, so that each subclass is free to determine its value any way it wants. *Don't* try to directly access the `value` attribute.

Why? A subclass may want to redefine how its value is computed. Using `getValue()` obeys the principle "*Program to an interface, not to an implementation.*"

2. Modify Valuable to *extend* Comparable

Every Valuable object *should* be comparable by value: a 10-Baht coin is worth less than a 50-Baht Banknote. So it makes sense that *Valuable* itself should be *Comparable*.

2.1 Modify *Valuable* to declare it is also *Comparable*.

2.2 Delete "... `implements Valuable`" from Coin and Banknote.

2.3 Implement compareTo in AbstractValuable. Order items by: (a) currency (so items with same curency are grouped together), and if currency is the same then (b) order by value. Be careful how you do the comparison. Some items may not have a currency. We may have some kinds of *Valuable* with very small value (0.000001) or very large value (1,000,000,000).

3. Refactor the `value` and `currency` attributes

Can you move `value` and `currency` to the superclass? Should they be private or protected? What should you do with getValue()?

4. Verify Your Code

Check your code:

- Coin and Banknote do not have equals or compareTo, they use the methods from *AbstractValuable*.
- The Purse and user interface do not depend on *AbstractValuable.* They depend only on Valuable. The word "AbstractValuable" should not appear anywhere in Purse.java!

Refactoring in Eclipse

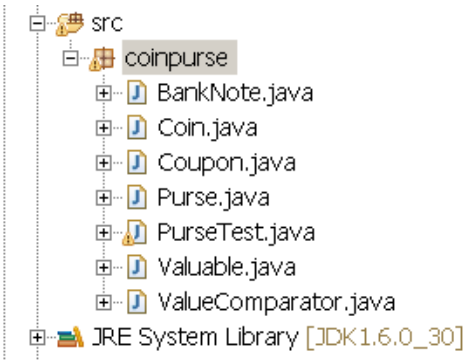
"*Refactoring*" means to restructure your source code. Eclipse and NetBeans have many refactoring operations to save time & reduce errors. The "Extract Superclass" refactoring creates a superclass and can move methods to the superclass.

We want to create an abstract superclass for Banknote and Coin.

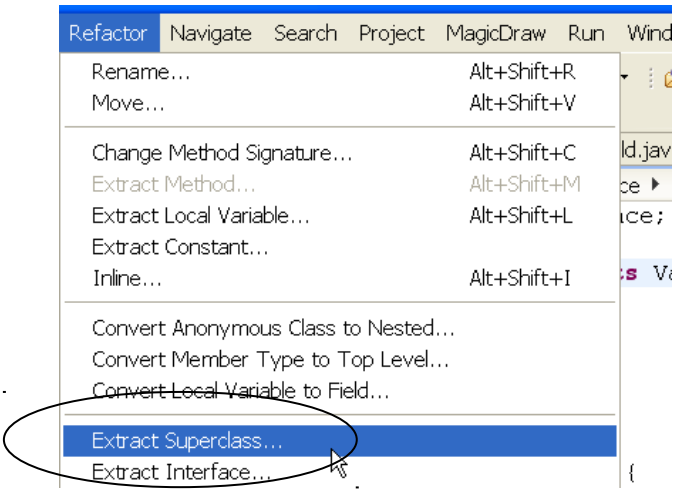
Eclipse can create an AbstractValuable superclass for you. Follow these steps:

1. Open one of the classes in the Eclipse editor.

Double-click on Coin to edit it.



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

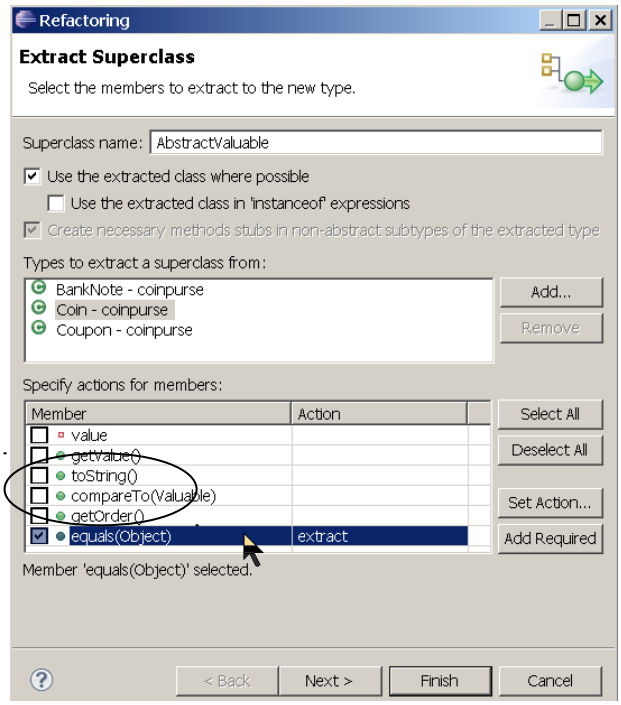


3. In the Extract Superclass dialog, enter `AbstractValuable` as the Superclass name.

4. Click Add... and add other classes you want to refactor (Banknote).

5. Select equals(Object) as the method to extract to the superclass.

Click the Next> Button.

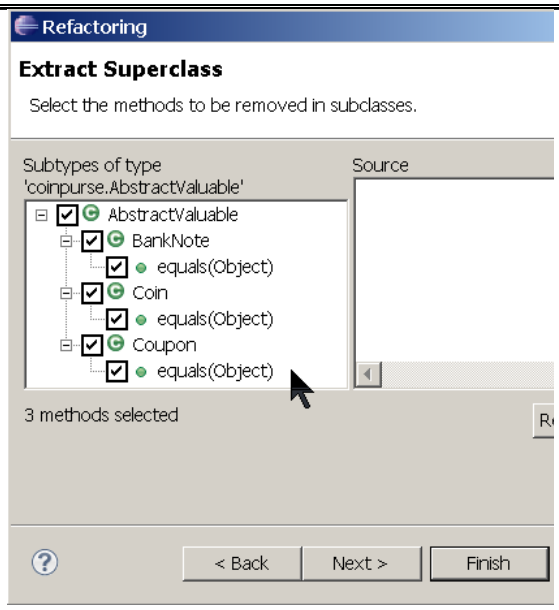


6. This dialog shows which classes will be subclasses and what methods will be extracted to AbstractValuable.

Check equals method for all 3 subclasses.

Click Finish.

You can ignore warning message about problems. You can fix these problems after refactoring.



How to Extract More Methods to Superclass

You can move methods from a subclasses to a superclass. In the Refactor menu choose "Pull Up".

Undo Refactoring

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