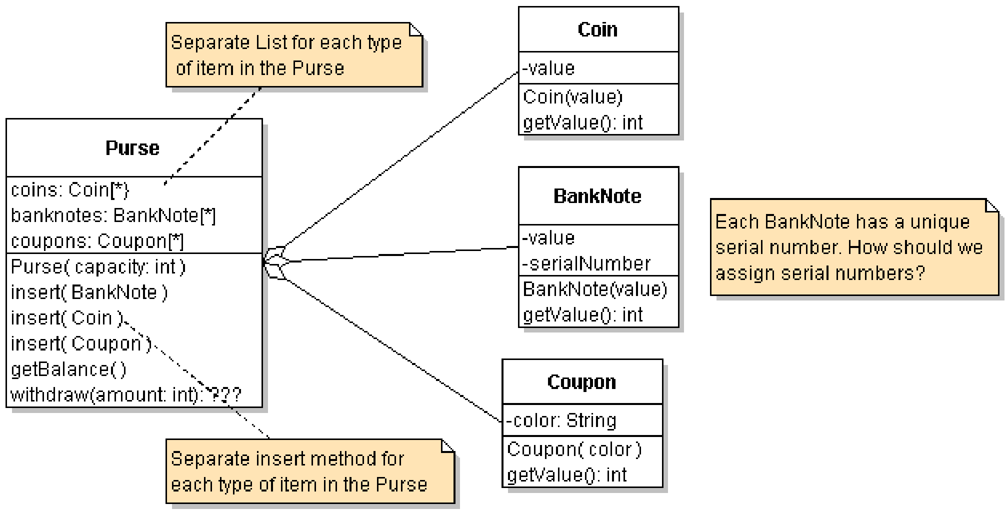
Objectives	Practice using an interface to enable your application to use polymorphism.
What to submit	Commit the revised code to the coinpurse project on Bitbucket. Ask TA to review UML diagram on paper.

The Coin Purse in Lab 2 is fine for storing coins; can we use it to store other kinds of money, too?

A Bad Design

We could add a separate List for each kind of money, including Coupons from KU Fair. We also need a separate `insert` method for each kind of money, like this:



This design might work (for now), but it is too complicated. And lots of duplicate code to handle different kinds of money.

A Polymorphic Purse

To simplify this, we need a way that the Purse can treat all kinds of money the same way.
We need to "abstract" the essential characteristic of money.

Question: What is the *essential characteristic* of money?

What does a Purse need to know about a Coin? A BankNote? Other money?

Answer: ____

The Purse doesn't care *how* a Coin or Banknote determines its value. The Purse only needs to know a ***how* to *ask*** the object for its **value**. That's the ***interface*** of money.

If we had a common ***interface*** for all kinds of money, the Purse could ask ***any*** money object for its value. The Purse would not know **what kind** of money it contains – just the interface to using money.

Preliminary: Create a git tag for your Lab2 Coin Purse project

A Git tag is a name you attach to a git revision, with an optional message. Tags act as bookmarks so you can checkout or go back to a particular revision of your code at any time. Make a tag named "lab2" to bookmark your solution to Lab 2.

```
> git tag -a lab2 -m "Solution to Lab 2 purse"
> git tag
```

(list of tags)

To checkout the code for a particular tag:

```
> git checkout lab2
```

To checkout the head revision of master branch (probably your most recent work)

```
> git checkout master
```

Exercise 1: Define a `Valuable` Interface

Create a Java interface named **`Valuable`** that defines a single method: **`getValue()`**.

An interface is a *specification* for a behavior, therefore interfaces should have good documentation! Write useful, complete Javadoc comments for the interface and the getValue() method.

```
package coinpurse;
//TODO write a useful Javadoc comment. What is this interface for?
public interface Valuable {
     //TODO document what getValue is supposed to do.
     public double getValue( );
}
```

The *real* money classes (Coin, Coupon, BankNote) 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 2: Define `Coin`, `Coupon`, and `BankNote` that implement Valuable

2.1 Modify Coin so that it implements Valuable. Coin should have these methods:

`getValue( )`	return value of this Coin.
`equals( Object obj )`	return true if obj is a Coin and has the same value as this coin.
`toString()`	return String description of Coin, such as "**`5-Baht coin`**".

2.2 Write a Coupon class that has a color and a value. The constructor determines the value using the color: Red = 100 Baht, Blue = 50 Baht, Green = 20 Baht.

Hint: Consider using an **`enum`** or **`Map`** for colors, instead of a bunch of "if" or "case" statements.

Coupon has these methods:

`getValue( )`	return the value of this coupon.
`equals( Object obj )`	return true if **`obj`** is a Coupon and has the same color as this Coupon -- ignoring case. "red" is the same as "RED".
`toString()`	return String describing the Coupon, such as "**`Red coupon`**".

2.3 Write a BankNote class. BankNotes can have any value (but our UI only creates 50, 100, ...). The BankNote constructor should assign each Banknote a unique serial number, starting from 1,000,000.

`getValue( )`	return the value of this BankNote.
`equals( Object obj )`	return true if **`obj`** is a BankNote and has the same value as this.
`toString()`	returns "**`xxx-Baht Banknote [`*`serialnum`*`]`**"

2.4 How to make serial numbers unique?

In the BankNote class define a static variable named nextSerialNumber and static method getNextSerialNumber() to return a unique serial number.

The BankNote constructor calls getNextSerialNumber to initialize serialNumber of a new BankNote.

Exercise 3: Draw a UML class diagram

Draw a UML class diagram showing the relationship between Coupon, Coin, BankNote, Valuable, and Purse. *Hint:* the purse does not depend on Coin.
Use separate paper for this.

Exercise 4: Write a Polymorphic Purse

Modify the Purse class so that deposit() and withdraw() accept anything that is Valuable. Change "Coin" to "Valuable" everywhere. "Refactor" to change the "coins" list to a better name, like money.

Programming Notes:

1. You can declare a **`List`** of objects using an Interface as the type parameter. For example:

```
List<Valuable> money;  // objects that implement Valuable
```

2. You can also *create* a collection or array using an Interface type. Examples:

```
money = new ArrayList<Valuable>( );
Valuable[] array = new Valuable[20];
```

3. The *Valuable* interface does not extend *Comparable* (no "compareTo" method), so you can't use Collections.sort (not yet). Comment out the Collections.sort(...). There are two alternatives:

(a) use a `Comparator` for sorting (see below).

(b) whenever you insert a Valuable into the Purse, add it to the list in order of increasing value. Then the list will always be sorted!! You won't need to use 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!

Write a PurseTest class to test your code.

5.1 Write one method for each test. You can use any Java code you like for testing; if you know how to use JUnit then use it.

5.2 If you are *not* using JUnit, then each test method should print a message *before* the test and another message *after* the test to say whether the test passes or fails.

5.3 Write a main method should invoke the test cases.

Example of Testing a Program in BlueJ

```
> Purse purse = new Purse( 5 );      // purse with capacity 5 Valuables
> Coin coin1 = new Coin( 10 );
> Coupon c   = new Coupon("blue");
> purse.insert( coin1 )
true
> purse.insert( c )
```

```
true
> purse.getBalance()
60
> purse.count()
2
> Valuable[] stuff = purse.withdraw(60);
                                      // withdraw returns array of 2 items
> stuff[0].toString()
"blue coupon"
> stuff[1].toString()
"10-baht coin"
> purse.getBalance()
0
```

Exercise 6: Write a **Comparator** for Valuable

How can we sort a List containing Coin, Coupon, and Banknote? We could make each class *implement Comparable* and write a compareTo method. But that would mean writing 3 methods that are almost identical.

Instead, we can write a separate *Comparator* and use it as a parameter in Collections.sort:

```
Collections.sort( list, comparator );
```

list of objects to sort

Comparator for comparing objects in list

Comparator is an interface with a type parameter (just like Comparable). It has only 2 methods:

```
public interface java.util.Comparator<T> {
    /**
     * Compare a and b.
     * @return < 0 if a should be "before" b,
     *         > 0 if a should be "after" b,
     *         = 0 if a and b have same lexical order.
     */
    public int compare(T a, T b);

    // writing equals is recommended but not required
    public boolean equals(Object obj);
}
```

6.1 Write a ValueComparator that compares any two objects of type Valuable. It should order the objects in a way that you want for sorting.

```
public class ValueComparator implements Comparator<Valuable> {
     public int compare(Valuable a, Valuable b) {
          // compare them by value.  This is easy.
     }
}
```

6.2 Write a test class to verify that ValueComparator can correctly sort a collection containing Coins, Coupons, and BankNotes. Use `Collections.sort( list, comparator )` for sorting.

6.3 In the Purse create a ValueComparator object. Since we only need one Comparator and it never changes, you can declare it as a final attribute:

```
     private final ValueComparator comparator
                    = new ValueComparator();
```

6.4 In the withdraw method, use the ValueComparator to sort the items in purse.