Coin Purse

Objectives	Implement an object-oriented program using a List for collection of objects.
Sample Code	Sample source code is coinpurse-sample.zip in the week3 folder.
What to Submit	Submit a project named coinpurse on Github. The source code should be in the package name coinpurse (which will appear *inside* your src/ directory) and there should be a descirptive README.md. Please **don't** submit binaries (use .gitignore as instructed in week1).

Requirements

1. Write an application to simulate a coin purse that we can **insert** and **remove** coins.

2. A purse has a **fixed capacity**. Capacity is the maximum *number* of coins that you can put in the purse, not the *value* of the coins. The value is unlimited.

3. A purse can tell us **how much money** is in the purse.

4. We can **insert** and **withdraw** money. For withdraw, we ask for an **amount** and the purse decides which coins to withdraw.

Application Design

In designing an O-O software application you need to do the following (among other things). Try to complete steps 2-4 on your own before reading the rest of the lab sheet.

1. Identify Classes: We need at least 3 classes for the application
   - Coin
   - Purse (if Purse seems quaint, think of it as a coin machine)
   - User Interface
2. Identify Responsibilities. What is the *main responsibility* of each class?
3. Assign behavior to classes: what methods should an object have to fulfill its responsibilities?
4. Determine attributes of objects: what does each object need to *know*?

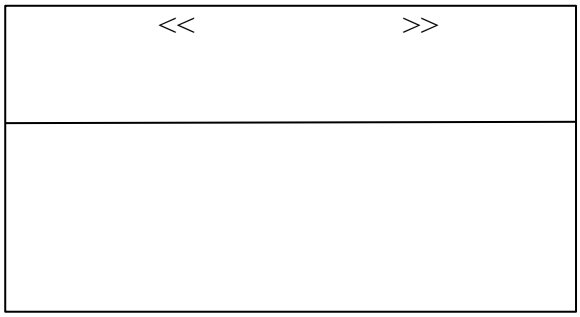
User Interface

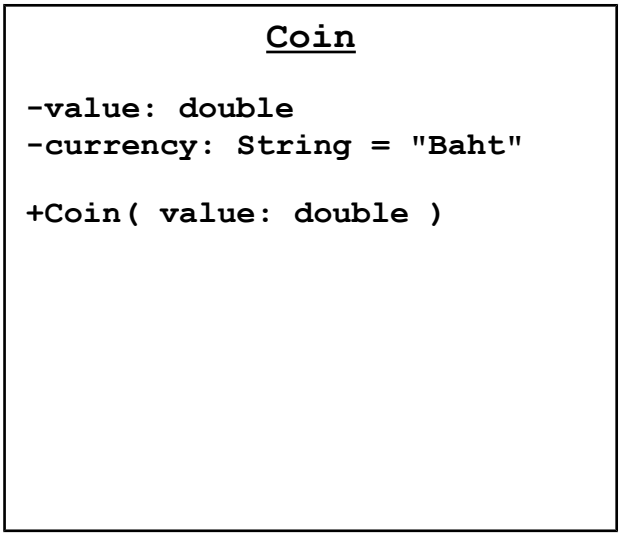
Coin object Purse object

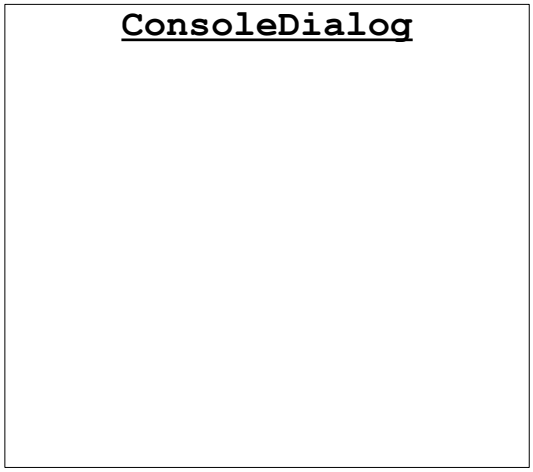
Exercise 1: Complete the UML diagram and submit it before end of lab

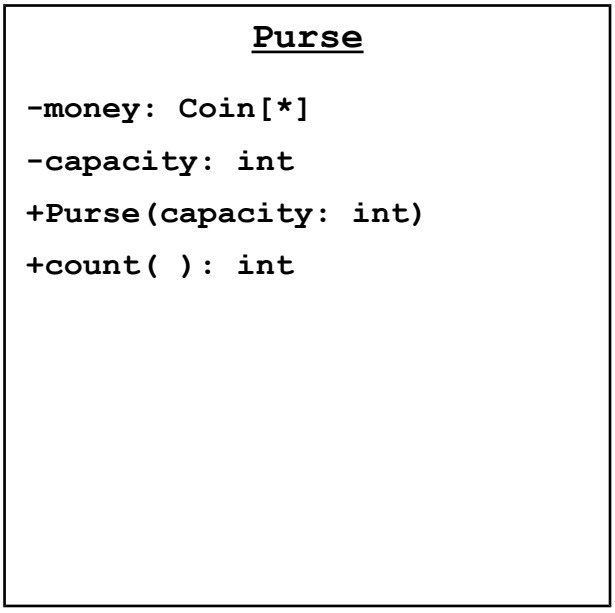
1. complete attributes and methods.
2. **show relationships**: association, dependency, implements.

The TAs must check your UML diagram before end of lab. You will be given one chance to fix errors.









Coin[*] means a collection of Coin objects.

Exercise 2: Design and Write the Coin Class

A Coin has a value and a currency that cannot be changed. A Coin can be **compared** to **other Coins**, so we can sort them by value. In this lab, set the default currency to "Baht". We will use other currencies later.

Behavior (doing):

get the value and currency

compare to another Coin (for sorting)

test for equality of two Coin objects

describe itself (toString)

Attributes (knowing):

A coin needs to know its value and currency

1. Implement the Coin class in a package named **`coinpurse`**.

2. Implement these methods, as described in the handout "*Fundamental Java Methods*".

`getValue( ), getCurrency()` accessor methods

`equals(Object arg)` two coins are equal if they have the same value and currency. Use the standard template for writing equals: test for null, test the class of the arg, then cast and compare values.

`compareTo` compare Coins, so that the *smaller* value comes first.

`coin1.compareTo( coin2 )`
- < 0 if `coin1` has less value than `coin2`
- = 0 if `coin1` and `coin2` have same value
- > 0 if `coin1` has greater value than `coin2`

3. Write good Javadoc comments for the class and all methods.

Write descriptive comments!

```
package coinpurse;
/**
 *  Coin represents coinage (money) with a fixed value and currency.
 *  @author Bill Gates 6010540000 (that's his net worth, not his id)
 */
public class Coin implements Comparable<Coin> {
```

`Comparable` is an interface in the Java API. Don't write this Interface yourself !!

2.2 Test the Coin class in BlueJ or Eclipse.

Test all the `Coin` methods. Here are some *examples*, but don't just copy! Create your own tests.

```
> import coinpurse.Coin;
> Coin one = new Coin(1);
> Coin five = new Coin(5);
> one.toString()
"1 Baht"
> one.compareTo(five)
-4        // 1-Baht comes "before" 5-Baht. Any value < 0 is ok.
> five.compareTo(one)
          // what should value be?
> one.compareTo(one)
          // what should value be?
> one.equals(five)
false
more tests...
```

Exercise 3: Implement the Purse Class

The sample code for this lab contains a partial Purse class.

1. Complete all the methods.
2. Write good Javadoc comments for class and methods.

Attributes (knowing):

know the capacity (how many coins it can hold)

know what objects are in the Purse.

Methods (behavior) and constructor:

Purse(capacity)	a constructor that creates an empty purse with a given capacity. `new Purse( 6 )` creates a Purse with capacity 6 coins.
int count( )	returns the *number* of coins in the Purse
double getBalance( )	returns the *value* of all the coins in the Purse. If Purse has two 10-Baht and three 1-Baht coins then getBalance() is 23.
int getCapacity( )	returns the capacity of the Purse
boolean isFull( )	return **`true`** if the purse is full
boolean insert(Coin)	Insert a coin in Purse. Returns **`true`** if insert OK, **`false`** if the Purse is full or the Coin is not valid (value <= 0).
Coin[ ] withdraw(amount)	try to withdraw money. Return an array of the Coins withdrawn. If purse can't withdraw the exact amount, then return **`null`**.
toString( )	return a String describing what is in the purse.

Example: A Purse with capacity 3 coins.

```
Purse purse = new Purse( 3 );
purse.getBalance( )          returns 0.0      (nothing in Purse yet)
purse.count( )               returns 0
purse.isFull( )              returns false
purse.insert(new Coin(5))    returns true
purse.insert(new Coin(10))   returns true
purse.insert(new Coin(0))    returns false. Don't allow coins with value <= 0.
purse.insert(new Coin(1))    returns true
purse.insert(new Coin(5))    returns false because purse is full (capacity 3 coins)
purse.count( )               returns 3
purse.isFull( )              returns true
purse.getBalance( )          returns 16.0
purse.toString()             returns "3 coins with value 16.0"
purse.withdraw(12)           returns null. Can't withdraw exactly 12 Baht.
purse.withdraw(11)           return array: [Coin(10), Coin(1)]
                             (coins can be in any order in array)
purse.getBalance()           returns 5.
```

3.2 Test the Purse

Test all the methods. Test both valid and invalid values, such as a coin with negative value. Also test *borderline cases*, like a Purse with capacity 1.

Hints for withdraw method

1. When you are trying to withdraw money, sort the coins first (or write an insert method that always keeps the coins sorted). Examine each coin starting from most valuable coin, and select any coin that will help you withdraw the amount needed. Each time you select a coin for withdrawal, deduct its value from the amount you need to withdraw. If the amount is reduced to zero, then withdraw succeeds!

While testing the coins for withdraw, you don't know if you can withdraw the exact amount yet. So, you have two choices:

a) remove coins from the Purse's money list as you go. But, if withdraw fails you must add them back to the Purse's money list.

b) *copy* a *reference* to coins you want to withdraw to a temporary list, but don't remove them from Purse's money list. Once withdraw succeeds then use the temporary list to remove those coins from Purse. If withdraw fails, you don't have to do anything since you didn't change the Purse's money list.

2. Don't use money.removeAll(templist) to remove coins from the Purse, because removeAll() will remove *all* coins that match (using equals) any Coin in templist. Instead, use a loop and remove one coin at a time. Use the list.remove(Object) method.

3. ArrayList has a method named toArray that copies elements of a list into an array:

```
Coin [] array = new Coin[ templist.size() ]; // create the array
templist.toArray(array);                     // copy to array
toArray also returns a reference to the same array
```

Exercise 4: Console User Interface

The sample code contains a *boring* **ConsoleDialog** that lets you deposit and withdraw coins.

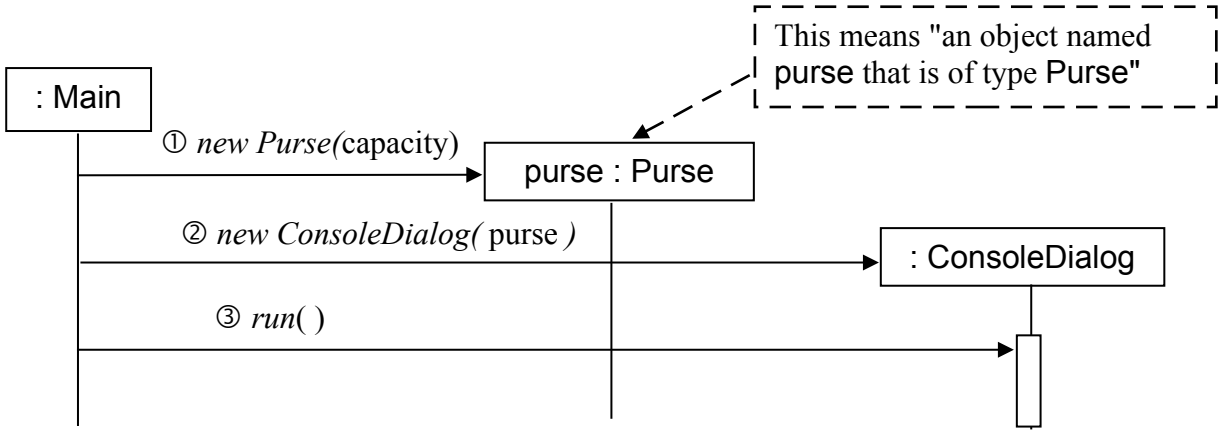
4.1 The ConsoleDialog needs a *reference* to a Purse. We **don't** want the user interface to *create* its own Purse! We want it *use an existing Purse*, not create one.

Modify the ConsoleDialog so that code (executed in the Main class) works:

```
// set a reference to purse object in the ConsoleDialog
ConsoleDialog ui = new ConsoleDialog( purse );
```

Exercise 5: Write a Main (Application) class to start the program

Write a **Main** class with a static **main** method to 1) create objects, 2) "connect" them together, and 3) invoke the user interface. The **main** method should implement this *Sequence Diagram*:



(1) create a Purse object with some capacity

(2) create a user interface and give it a *reference* to the purse it should use.

(3) call `consoleDialog.run( )` to start the ConsoleDialog object

```
package coinpurse;
```

```
/**
 * Main (application) class creates objects and starts the application.
 */
public class Main {
   private static int CAPACITY = 10;

   public static void main( String [] args ) {

   //TODO implement the steps shown in the sequence diagram
   }
}
```

OO Design Principle: Dependency Injection

Objects need to interact with each other, so we need a way for many objects to "know about" or share a reference to an object they interact with. We also want to write flexible, reusable code and enable polymorphism.

Both of these factors mean we can't just write "new Purse()" or "new GuessingGame()" whenever we want to use an object. A solution is to give one class responsibility for creating shared objects, and then set a reference into other objects. This is called *dependency injecttion.*

Here are examples of 2 ways of doing it:

1. Inject a reference using constructor (called *constructor injection*):

```
// in the main class: GuessingGame is the shared object
GuessingGame game = new GuessingGame(1000);
// "set" the game to play into the user interface
GameConsole ui = new GameConsole( game );
```

2. Inject a reference using a set method (called *setter injection*):

```
// in the main class:
GuessingGame game = new GuessingGame(1000);
// create user interface without any game yet
GameConsole ui = new GameConsole( );
// "set" the game to play
ui.setGame( game );
```

Optional: Write a method to view contents of Purse

For testing, we'd like to see exactly what is in the Purse. Write a method like this:

`List<Coin> getContents( )`	return the the contents of the purse as an **immutable** List<Coin>. The List cannot be modified.

We must ensure that someone cannot use getContents() to surreptiously modify the purse, like this:

```
// take a coin from the purse without using withdraw!
Coin stolen = purse.getContents().remove( 0 );  // steal first coin!
```

The way to prevent this is the make the List of coins be *immutable* or *unmodifiable*. The java.util.Collections class has a method to do this, named unmodifiableList( ).

It creates a *view* that "wraps" the original list but prevents using methods that modify the List. This is a *view,* not a copy (can you write some code to prove it is not a copy?). The *view* is a List and implements all the methods of List, but prevents the contents being modified.

List methods used in this Lab

Create ArrayList that can hold any Object	`List list = new ArrayList( );` `// List is an interface, ArrayList is a class.`
Create an ArrayList to hold Coin objects	`List<Coin> money = new ArrayList<Coin>( );`
Number of items in a list	`int size = money.size(); // # items in the list`

Add object to a list.	`boolean ok = list.add( object );`
	`if ( ! ok ) /* add failed! */;`
	`  // This never happens for ArrayList`

Get one Coin from list without removing it.	`Coin coin = money.get(0);  // get item #0`
	`Coin coin2 = money.get(2); // get the 3rd item`

Get one Coin and remove it from list	`Coin c = money.remove(0); // remove item 0`
	or:
	`Coin coin = money.get(k); // get some coin`
	`money.remove(coin);  // remove matching coin`
	Note: **money.remove(somecoin)** uses the **equals()** method of **Coin** to find the first object in the list that equals **somecoin**. The object removed may not be the same object as **somecoin**!

Iterate over all elements in a list	`// A for-each loop to print each coin in list:`
	`for(Coin coin : list)`
	`  System.out.println( coin );`
	`// A for loop with an index (k).`
	`for(int k=0; k < list.size(); k++)`
	`  System.out.println( list.get(k) );`

Copy a List into an array of exactly the same size	`List<String> list = new ArrayList<String>( );`
	`// first create array of the correct size`
	`String[] array = new String[ list.size() ];`
	`list.toArray( array ); // copies list to array`

Copy everything from list2 to the end of list1.	`List list1 = new ArrayList( );`
	`List list2 = new ArrayList( );`
	`list2.add( ... ); // add stuff`
	`list1.addAll( list2 );  // copy everything`