Coin Purse

Objectives	Implement an object-oriented program using a List for collection of objects.
Sample Code	Sample source code is in the Github Classroom project for this course. You can also view sample code at https://github.com/OOP2018/coinpurse-start/
What to Submit	Create a project using the Github classroom URL given in class, and do the assignment. Update the README.md with information about your project. Push your completed code back to Github. Your repository URL will be of the form: `https://github.com/OOP2018/coinpurse-`*`yourlogin`*

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

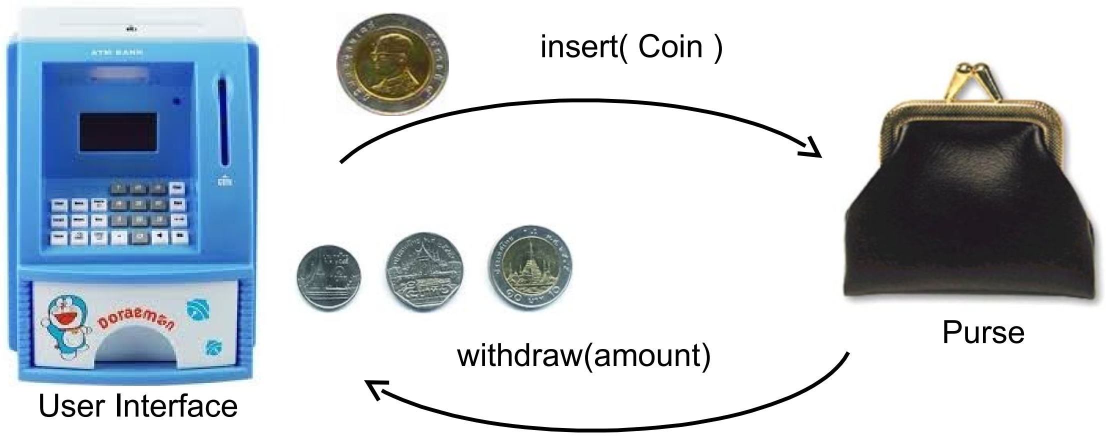
    Purse (like a Coin Machine)

    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*?



Problem 1: Implement the Coin Class

A Coin has a value and a currency that cannot be changed after the coin is created. A Coin can be **compared** to **other Coins. Coin** and other classes in this lab should be in the package `coinpurse`.

1. Implement the Coin class with these methods and constructor:

`Coin(double value, String currency)`	Constructor for a Coin with a value and currency. The value must not be negative but may less than 1, e.g. 0.25 Baht.

`double getValue( )`
 `String getCurrency( )`
	Accessor methods. Coin is *immutable*, so there are no "set" methods.

`boolean equals(Object arg)`	Two coins are equal if they have the same value *and* same currency. Use the standard template for writing equals, as in the *Fundamental Methods* doc.
`int compareTo(Coin coin)`	order Coins by value so that the coin with smaller value comes first. The meaning of compareTo is: `a.compareTo(b) < 0` if a has order *before* b `> 0` if a has order *after* b `= 0` if a and b have same order For now, compareTo ignores the currency value.
`String toString()`	return String such as "5-Baht" or "0.25-Rupie".

2. Declare that Coin implements the java.lang.Comparable interface, but can only be compared to other Coins:

```
public class Coin implements Comparable<Coin>
```

Comparable is an interface in the Java API. *Don't write the interface yourself!*

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

```
package coinpurse;
/**
 *  Coin represents coinage (money) with a fixed value and currency.
 *  @author Bill Gates
 */
public class Coin implements Comparable<Coin> {
```

1.2 Write sample code to demonstrate sorting (test your compareTo method)

The purpose of implementing Comparable is so we can sort a list (or array) of Coins. Here is an example:

```
// Don't just copy this code.  Write your own test code.
List<Coin> coins = new ArrayList<Coin>( );
coins.add( new Coin(10.0, "Baht") );
coins.add( new Coin(0.5, "Baht") );
coins.add( new Coin(2.0, "Baht") ); // the most hated coin
coins.add( new Coin(1.0, "Baht") );
printCoins( coins );    //TODO write a method to print the coins
// This static method sorts any list of Comparable objects
java.util.Collections.sort( coins );
printCoins( coins );  // the coins should be sorted by value now
```

Write a class named MoneyUtil with a sortCoins method to demonstrate that your compareTo is correct and you can sort the coins by value. It is needed by the coin purse. Don't just copy this code. Write a method to print the coins, and create more coins that just 4 coins. What if 2 coins have same value but different currency? Try it.

Problem 2: Implement the Purse Class

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

1. Complete all the methods 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.
double getBalance( )	returns the *value* of all coins in the Purse. If Purse has two 10-Baht and three 1-Baht coins then getBalance() is 23. Ignore the currency for now.
int count( )	Return count of number of items in the Purse.
int getCapacity( )	returns the capacity of the Purse
boolean isFull( )	return **`true`** if the purse is full, false if not full.
boolean insert(Coin)	Insert a coin into Purse. Returns **`true`** if insert OK, **`false`** if the Purse is full or the Coin is not valid (value <= 0).
Coin[ ] withdraw(double 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.isFull( )`	returns false
`purse.insert(new Coin(5,"THB"))`	returns true
`purse.insert(new Coin(10,"THB"))`	returns true
`purse.insert(new Coin(0,"THB"))`	returns false. Don't allow coins with value <= 0.
`purse.insert(new Coin(1,"THB"))`	returns true
`purse.insert(new Coin(5,"THB"))`	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)

2.2 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.

2.3 (Optional) Test using JUnit and PurseTest

The sample code has a source file named PurseTest.java. This is a JUnit 4 test class. To use this class you must do 2 things:

1. add JUnit 4 to your project (Eclipse will prompt you to do this)
2. run the tests by right-click on PurseTest (in Project pane) and choose Run As -> JUnit Test.
3. Eclipse will show which tests "pass" or "fail". For failed tests, it shows where the code failed.

Hints for withdraw method

1. When you want to withdraw an amount, sort the coins first. Examine each coin starting from most valuable coin. If the value of this coin is less than or equal the amount you need to withdraw, then copy the coin reference to a temporary list (of stuff you want to withdraw).
Each time you choose a coin for withdrawal, deduct its value from the amount you still need to withdraw. If the amount is reduced to zero, then withdraw succeeds!

While you are choosing which coins to withdraw, you don't know if you can withdraw the exact amount yet. So, you have two choices:

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

b) 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. I think this choice involves more chance for error, so I suggest a).

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

3. Withdraw needs to return an array of Coins (not a list). 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 3: Console User Interface

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

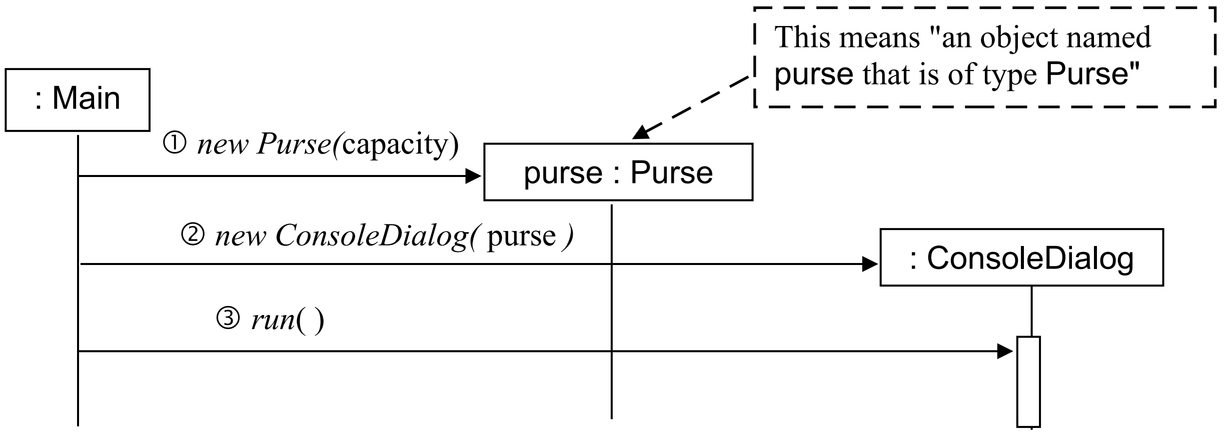
The ConsoleDialog should work as-is, but if your Coin or Purse class differs from the specification there will be errors. Try to fix your Coin or Purse so there are no errors.

To use the ConsoleDialog you have to *give* it a reference to a Purse. It doesn't create its own purse.

```
ConsoleDialog ui = new ConsoleDialog( purse );
ui.run();
```

Exercise 4: Complete the Main (Application) class to start the program

Complete the **Main** class (in starter code) with a static **main** method to 1) create objects, 2) "connect" them together, and 3) invoke the user interface. The **main** method implements this *Sequence Diagram*:



This means "an object named purse that is of type Purse"
: Main
① new Purse(capacity)
purse : Purse
② new ConsoleDialog(purse)
: ConsoleDialog
③ run( )

(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.

Exercise 5. Write MoneyUtil method to Practice using Lists

Write a class name MoneyUtil in the coinpurse package. Complete 2 methods, plus a main() method to invoke them and print the results.

static List<Coin> filterByCurrency(List<Coin> coins, String currency)	Return a List of Coins that contains only the coins from coins (the parameter) that have same currency as the currency parameter.
static void sortCoins(List<Coin> coins)	Sort a list of coins and print the result on the console. Write a separate method to print the list.

OO Design Principle: Use Dependency Injection to Connect Objects

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

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 have this class set a reference into other objects. This is called *dependency injection.*

Here are 2 ways of doing dependency injection:

1. Set a reference using constructor (called *constructor injection*).
In this example, the ConsoleDialog needs a reference to the Purse, so we give it a Purse object as parameter to the constructor.

```
// in the main class: Purse is the shared object
Purse purse = new Purse(10);
// "set" the purse to use into the UI
ConsoleDialog ui = new ConsoleDialog( purse );
```

2. Set a reference using a set method (called *setter injection*). Not used in this lab.

List methods used in this Lab

Create an ArrayList to hold Coin object.	`List<Coin> money;`
	`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! */;`
	`     // add never fails for ArrayList`

Get one Coin from list without removing it.	`Coin coin = list.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`
	<u>or:</u>
	`Coin coin = money.get(k); // get some coin`
	`money.remove(coin);       // remove matching coin`
	<u>Note</u>: **`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**!

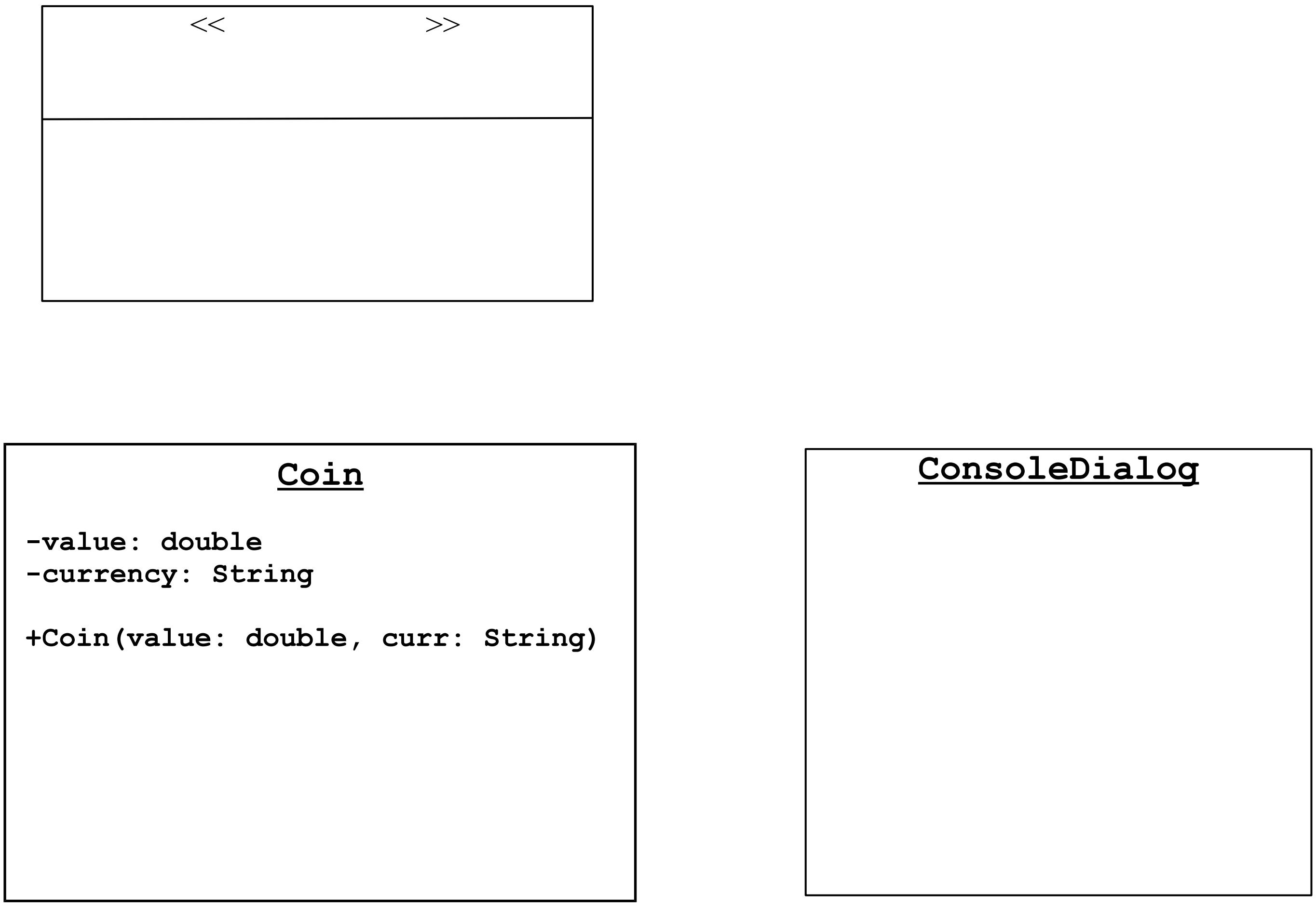
Iterate over all elements in a list	`// A for-each loop to print each coin in list:`
	`for(Coin coin : list)`
	`     System.out.println( coin );`
	`// A for loop with an index (k).`
	`for(int k=0; k < list.size(); k++)`
	`     System.out.println( list.get(k) );`

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`

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

1. Complete this UML diagram. Show the Coin, Purse, and ConsoleDialog classes, and interface used.
3. Ask a TA to check the diagram before leaving the lab today. Fix any errors.



<< >>
Coin
-value: double
-currency: String
+Coin(value: double, curr: String)
ConsoleDialog