

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 <i>inside</i> your src/ directory) and there should be a descriptive 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.

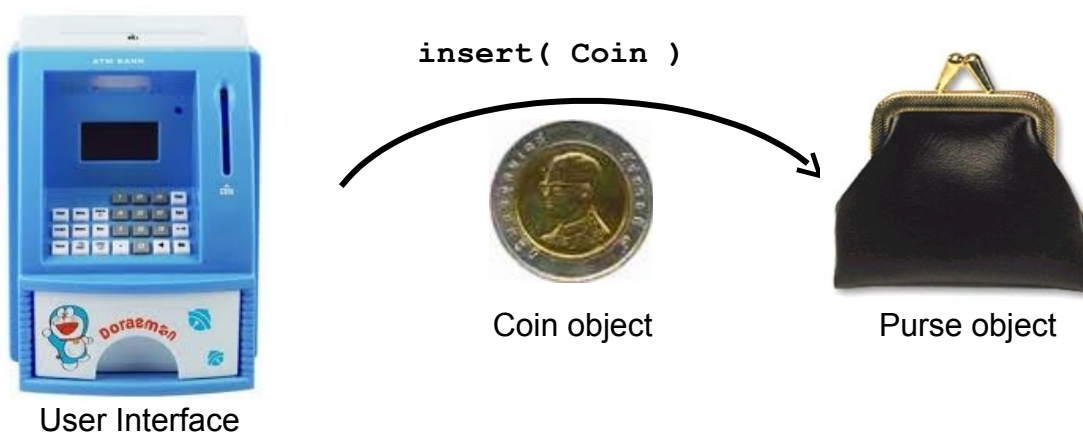
1. Identify Classes: We need at least 3 classes for the application

Coin

Purse (if Purse seems quaint, think of it as a coin machine)

User Interface

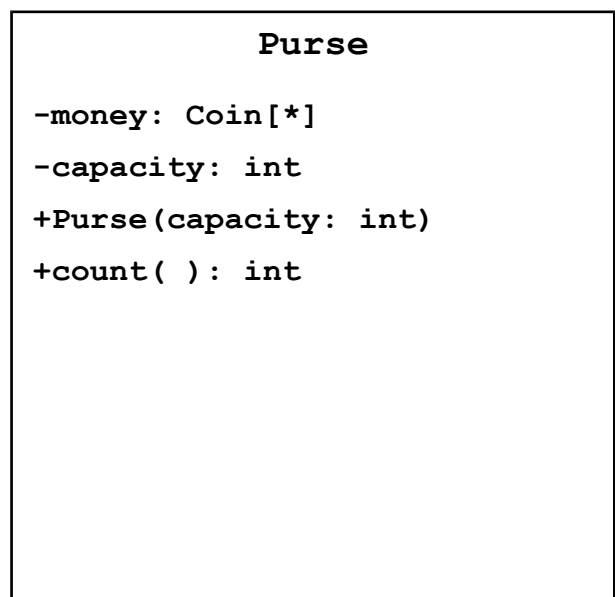
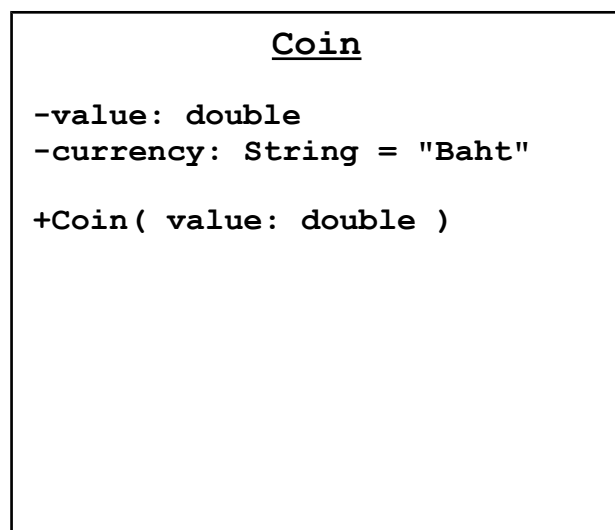
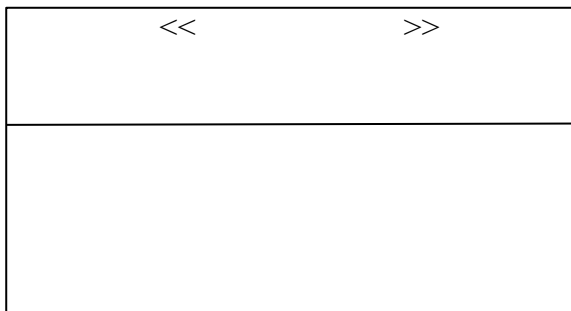
2. Identify Responsibilities. What is the *main responsibility* of each class?
3. Assign behavior to classes: what methods should an object have to fulfill its responsibilities?
4. Determine attributes of objects: what does each object need to *know*?



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.

```
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> {
```

Write descriptive comments!

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. <code>new Purse(6)</code> creates a Purse with capacity 6 coins.
int count()	returns the <i>number</i> of coins in the Purse
double getBalance()	returns the <i>value</i> of all the coins in the Purse. If Purse has two 10-Baht and three 1-Baht coins then <code>getBalance()</code> 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 <u>array</u> 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.

<code>Purse purse = new Purse(3);</code>	
<code>purse.getBalance()</code>	returns 0.0 (nothing in Purse yet)
<code>purse.count()</code>	returns 0
<code>purse.isFull()</code>	returns false
<code>purse.insert(new Coin(5))</code>	returns true
<code>purse.insert(new Coin(10))</code>	returns true
<code>purse.insert(new Coin(0))</code>	returns false. Don't allow coins with value <= 0.
<code>purse.insert(new Coin(1))</code>	returns true
<code>purse.insert(new Coin(5))</code>	returns false because purse is full (capacity 3 coins)
<code>purse.count()</code>	returns 3
<code>purse.isFull()</code>	returns true
<code>purse.getBalance()</code>	returns 16.0
<code>purse.toString()</code>	returns "3 coins with value 16.0"
<code>purse.withdraw(12)</code>	returns null. Can't withdraw exactly 12 Baht.
<code>purse.withdraw(11)</code>	return array: [Coin(10), Coin(1)] (coins can be in any order in array)
<code>purse.getBalance()</code>	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 you want to withdraw from the Purse's money list. But, if withdraw fails you must add them back to the 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 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
```

Exercise 3.3: Write a method to view contents of Purse

For testing and creating a graphical UI (later), 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 a List<Coin>.
--	---

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

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

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 "wraps" the list in a *view*, not a copy of the List. The *view* it creates behaves like a List, but cannot be modified (try it!).

Use this method in `getContents()` to prevent someone from stealing coins as in the example above.

Exercise 4: Console User Interface

For this lab, you can use the *boring* `ConsoleDialog` in the sample code. No coding necessary.

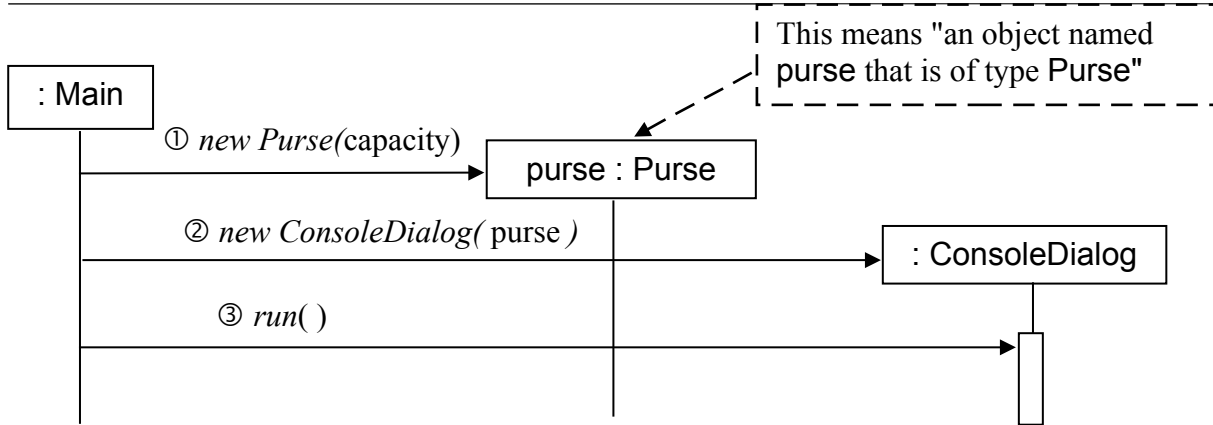
The `ConsoleDialog` needs a *reference* to the Purse. We **don't** want the user interface to *create* its own Purse! We want it *use our Purse*, not create one.

So, we *set a Purse reference* in the `ConsoleDialog` object using the constructor:

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

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
    }
}
  
```

Connecting Objects with *Dependency Injection*

The main (application) class sets (injects) the Purse reference in ConsoleDialog.

The ConsoleDialog uses a Purse but it does not create the Purse!

You did the same thing in the Guessing Game.

Giving one object a reference to another object like this is called **dependency injection**.

Dependency injection makes your code more flexible and reusable.

List methods used in this Lab

Create ArrayList that can hold any Object	List list = new ArrayList (); // List is an <i>interface</i> , ArrayList is a <i>class</i> .
Create an ArrayList to hold Coin objects	List <Coin> money = new ArrayList <Coin>();
Number of items in a list	int size = money. size (); // size of a 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 = coins. get (0); // get item #0 Coin coin2 = coins. get (2); // get item #2
Get one Coin and remove it from list	Coin c = coins. remove (0); // remove item 0 <u>or:</u> Coin coin = coins. get (k); // get some coin coins. remove (coin); // remove matching coin <u>Note:</u> coins.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 all to list1