Coin Purse

Objectives	Implement an object-oriented program using a **List** for collection of objects.
Sample Code	Sample source code for Purse and ConsoleDialog in week2 folder.
What to Submit	Create a project named CoinPurse and commit it to your Git repository on Bitbucket. Make the repository private and give write access to skeoop:ta .

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 <u>*number*</u> 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

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

    Coin

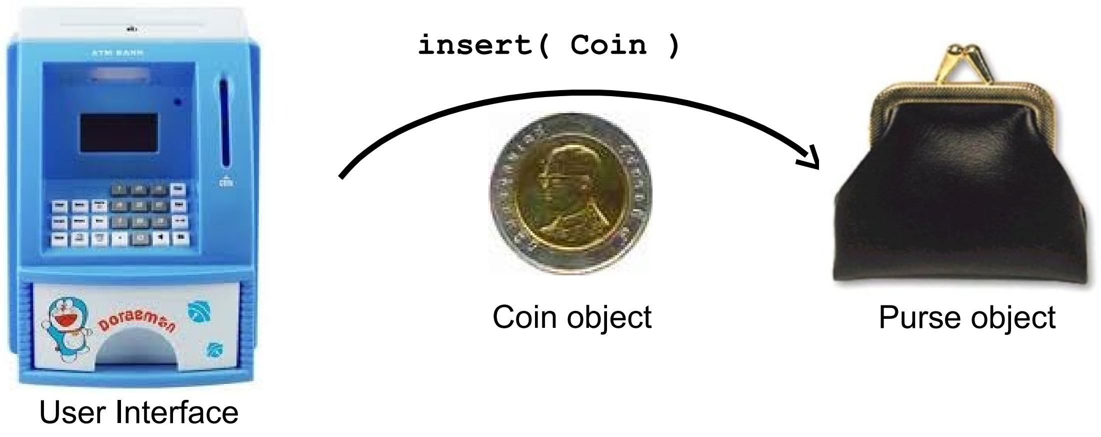
    Purse

    User Interface

2. Identify Responsibility. What is *main responsibility* of each class?

3. Assign behavior: what methods should an object have in order to fulfill its responsibilities?

4. Determine attributes: what does each object need to *know*?



Exercise 1: The Coin Class

A Coin has a value that cannot be changed. Coins can be **compared** to **other Coins**, so we can sort them by value.

Behavior (doing):

get the value (an *accessor* method)

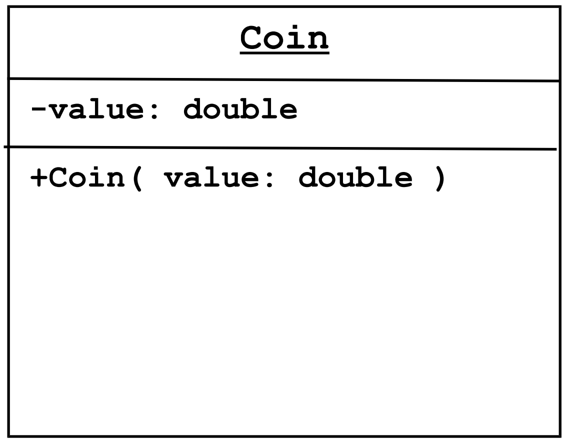
compare to another Coin

test for equality to another Coin

describe itself (toString)

Attributes (knowing):

A coin needs to know its value.



1.1 Complete the UML Diagram

There should be 4 methods and 1 constructor.

1.2 Implement Coin

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

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

`getValue( )`	get the value of this coin
`equals(Object arg)`	is true if a) arg is not null, b) arg is a Coin, c) arg has same value as this Coin. The parameter to `equals` must be declared as `Object`.
`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`

Design Issue: Why is there not a setValue( ) method in Coin?

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

```
package coinpurse;
/**
 *  Coin class represents a coin with a fixed value.
 *  @author Bill Gates 50540000
 */
public class Coin implements Comparable<Coin> {
```

Use this package name.

Write descriptive comments!

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

1.3 Test the Coin class in BlueJ. If you use Eclipse, then write your own tests.

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 2: 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.

Behavior:

insert and withdraw Coins

provide the balance (value) of stuff in the Purse

check if Purse is full

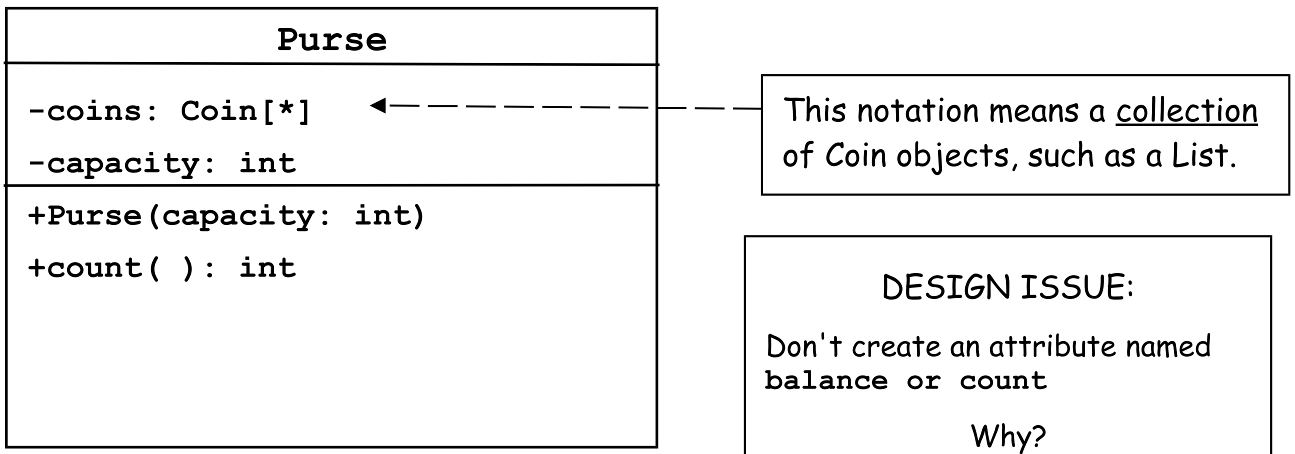
Attributes (knowing):

know the capacity (how many coins it can hold)

know what objects are in the Purse.

Methods:

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)	try to 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.



DESIGN ISSUE:

Don't create an attribute named **`balance`** or **`count`**

Why?

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(5))`	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 20.0
`purse.toString()`	returns `"3 coins with value 20.0", or "2 5-Baht coin, 1 10-Baht coin"`
`purse.withdraw(11)`	returns `null`. Can't withdraw exactly 11 Baht.
`purse.withdraw(15)`	returns an array: `[Coin(10), Coin(5) ]`
`purse.getBalance()`	returns 5.

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.

Hints for withdraw:

1. When you are trying to withdraw money, sort the coins and try to withdraw starting from the most valuable coin down to the least valuable coin. Don't remove coins from Purse's List while you are trying to withdraw. Instead, *copy* the coin *references* to a *temporary* list. Each time you add a Coin to the temporary list, deduct its value from the amount you need to withdraw. If the amount is reduced to zero then you have succeeded.

If you succeed, then use the temporary list to remove coins from the Purse's list of coins. If withdraw doesn't succeed, then just return false! The temporary list is a local variable so it will be destroyed when the method returns.

2. Don't use coins.removeAll(templist) to remove coins from Purse, because removeAll() will remove *all* coins that are equal (using equals) to 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: Console User Interface

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

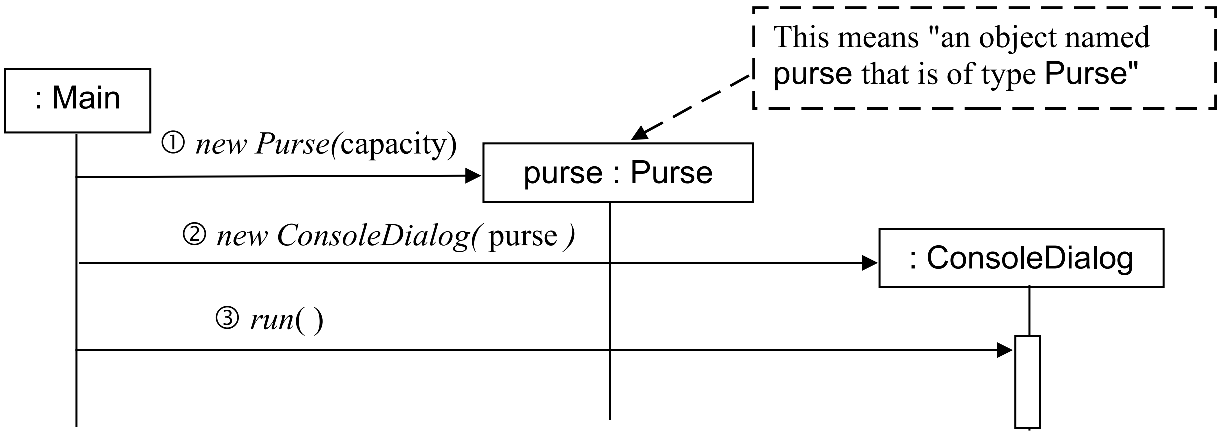
The ConsoleDialog needs a *reference* to the Purse. We **don't** want the user interface to *create* its own Purse! We want it *use our Purse*, not create one.

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

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

Exercise 4: Write a Main class (Application class) to create objects and start the program

Write a **`Main`** class with a static **`main`** method to create objects, "connect" them together, and start the program. 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

```
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 (1) create the purse
   //TODO (2) create ConsoleDialog and give it a reference to the purse
   //TODO (3) run ConsoleDialog
   }
}
```

The main method must be declared like this.

Connecting Objects with *Dependency Injection*

The main (application) class sets the Purse object in ConsoleDialog, so the ConsoleDialog just *uses* a Purse – it doesn't create the purse!

Setting a reference to one object into another object is called ***dependency injection***.

Dependency injection makes your code more flexible and reusable.

Extra: Graphical UI

If you finish early, write a graphical UI for using the coin purse. It should replace the ConsoleDialog but be invoked the same way – that is, you *inject* a Purse object using constructor, then call a run() method to show the UI.

List methods used in this Lab

Create an ArrayList that can hold anything	`List list = new ArrayList( );` `// List is an interface, ArrayList is a class.`
Create an ArrayList to hold Coin objects	`List<Coin> coins = new ArrayList<Coin>( );`
Number of items in a list	`int size = coins.size(); // size of a list`
Add object to a list.	`boolean ok = list.add( object );` `if ( ! ok ) /* add failed! */`
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` or: `Coin coin = coins.get(k); // get some coin` `coins.remove(coin);     // remove matching coin` Note: `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	`// Use for-each loop to print each coin in list:` `for(Coin coin : list)` `     System.out.println( coin );` `// Use 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`