Coin Purse

Objective	Practice using list methods to implement an O-O style application.
Starter Code	Accept the assignment on Github Classroom to create a repository with starter code. Assignment URL: https://classroom.github.com/a/DU03pnX3
What to Submit	Submit your code to Github. Please include a README.md file.

Requirements

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

2. A purse has a **fixed capacity**. The capacity is the *number* of objects that you can put in the purse. Capacity limits the *number* of objects (money), not the *value* of the objects.

3. A purse should tell us **how much money** is in the purse. Since money can have different currencies, this is a little complex. We have to ask for the balance using a specific currency.

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

User Interface

Coin object

Purse object

Application Design

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

   Coin - things we can put in the purse

   Purse - manages money

   User Interface - interacts with the user

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

3. Determine the behavior: what should each object *do*?

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

1. Coin

Coin has a value and currency. that cannot be changed.
Behavior:

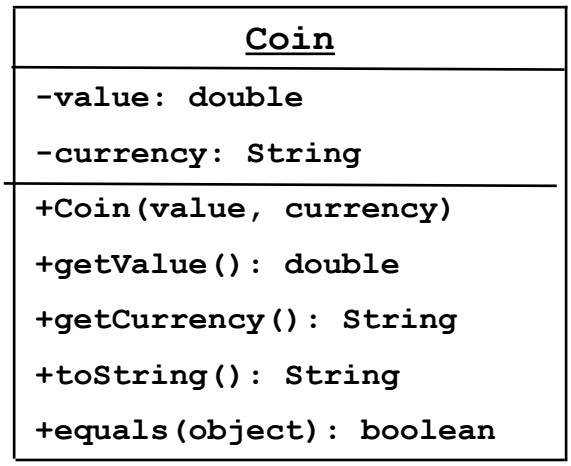
get the value and currency

test for equality to another Coin

describe itself (toString)

toString for Coins:

if the value is a whole number (1.0, 5.0, etc)
then return a string like "`100-Yen coin`",
otherwise, format it with 2-decimal places, e.g.:
"`0.50-Baht coin`" (we may improve this in another lab)



1.1 Implement Coin

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

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

`equals(Object arg)` is true if arg is a Coin and arg has same value and currency as this Coin.

`toString()` if the value of the coin is a whole number (1, 5, 10, etc) then return a String such as "5-Baht coin". Otherwise, return a string with 2-decimal places, e.g. "0.25-Baht coin". (We may improve this in another lab).

1.2 Test the Coin class in BlueJ, jshell, or test code in a file

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, "Baht");
> Coin five = new Coin(5, "Baht");
> one.toString()
"1-Baht coin"
> one.equals(five)
false
> Coin c = new Coin(1, "Baht");
> one.equals(c)
true
> c = new Coin(5, "Ringgit");
> five.equals(c)
false
```

2. Make Coins Comparable (for sorting or ordering)

The Purse (problem 4) will need to be able to **sort** coins by value, so define a method used for ordering coins. In Java, this is done by implementing an interface named *Comparable* that specifies one method named `compareTo`.

Comparable is part of the Java API. **Don't** write Comparable yourself, just use it in your code.

1. Declare that the Coin class implements *Comparable* for comparison to other coins.

```
package coinpurse;
/**
```

```
 *  Coin class represents a coin with a fixed value and currency.
 *  @author Bill Gates
 */
public class Coin implements Comparable<Coin> {

    //TODO define attributes
    //TODO define constructor and methods

    public int compareTo(Coin other) {
         // order coins by value - see below
    }
}
```

2. Write the `compareTo` method so that it compares coins by value.

`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`

The exact value returned by compareTo does not matter. The only requirement is that a negative value means coin1 is "before" coin2 in the sort order (called lexical order), a positive value means coin1 is "after" coin2, and return value of 0 means that they have the same sort order (for example, two 1-Baht coins).

3. Test it.

```
> Coin a = new Coin(5, "THB");
> Coin b = new Coin(2, "THB");
> a.compareTo(b)
1  (or any positive value)
> b.compareto(a)
-1 (or any negative value)
> b.compareTo(b)
0
```

4. Try sorting an array of coins.

```
> String bt = "Baht";
> Coin[] coins = { new Coin(5,bt), new Coin(1,bt),
                   new Coin(0.5,bt), new Coin(2,bt) };
> java.util.Arrays.sort(coins);
> coins
```

(if you inspect it, the coins should see the coins are sorted -- but there is an easier way)

```
> java.util.Arrays.toString( coins )
["0.50-Baht coin", "1-Baht coin", "2-Baht coin", "5-Baht coin"]
```

More practice: you can do this with an array of String or Double. Sorting strings can be surprising.

4. Order coins by currency, too. When we sort or search a List of coins, we would like all the "Baht" and "baht" coins together, then all the "Peso" coins together, then "Ringgit" coins, etc.

`coin1.compareTo( coin2 )`

- < 0 if `coin1` has a currency that is "before" coin2's currency in dictionary order (ignoring case), or if both coins have same currency and coin1 has smaller value
- = 0 if `coin1` and `coin2` have same currency (ignoring case) and same value.
- > 0 if `coin1` has greater currency or value than `coin2`

This is a challenge for you: write a compareTo that orders coins first by currency (ignoring case), then by value for coins of same currency. Note that String has a method compareToIgnoreCase.

```
> Coin tenbaht = new Coin(10,"Baht");
> Coin satang = new Coin(0.5, "Baht");
> Coin yen = new Coin(5, "Yen");
> tenbaht.compareTo(yen)
-24  (any negative value, since "Baht" is before "Yen")
> tenbaht.compareTo(satang)
1    (positive, since 10-Baht > 0.50-Baht)
> yen.compareTo(new Coin(100, "Yen"))
-1   (5-Yen < 100-Yen)
```

3. List Practice

Use the Java documentation for ArrayList and Collections (not Collection) to complete these exercises. You can put this code in a static method (easy to re-run), BlueJ codepad, or jshell.

```
import java.util.*;   // for List, ArrayList, and Collections

// (1) Create a list of coins.
List<Coin> list = new ArrayList<Coin>( );

// (2) add one coin and check the list
Coin twenty = new Coin(20, "Astra");
list.add( twenty );
list.size()
list.get(0)

// (3) add many coins. In BlueJ the loop must be on one line.
for(int val=9; val > 0; val--) list.add(new Coin(val, "Baht"));

// (4) display the size of the list in console window
System.out.println("List size is " + list.size() );

// (5) show what is in the list. Use a a "foreach" loop.
for(Coin c : list) System.out.println( c );

// (6) Sum the total value of coins (ignore currency)
_____________________________________________________________________
_____________________________________________________________________

// (7) sort the list.
java.util.Collections.sort( list );

// (8) Print the list again.  Should be sorted. (repeat (5))

_____________________________________________________________________

// (9) remove the twenty Astra coin from list using its index.
_____________________________________________________________________

// (10) remove everything from the list.

// (11) verify the list is empty -- two ways
list.size()
list.isEmpty()
```

4. The Purse

Write a Purse class that manages some coins.

Behavior:

insert and withdraw Coins

inquire the balance of Purse (by currency)

check if Purse is full

Attributes (what it knows):

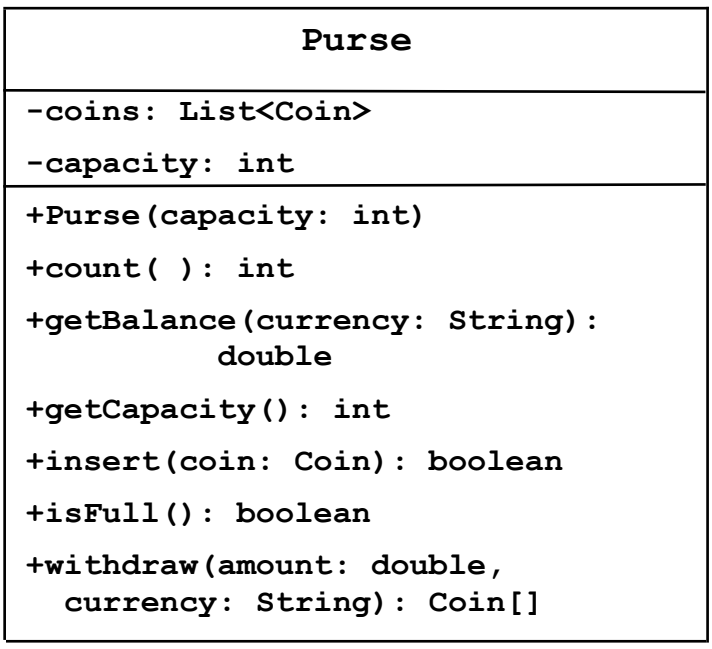
the capacity (how many coins it can hold)

the coins in the Purse

Methods:

Purse(capacity)	a constructor that creates an empty purse with a given capacity. `new Purse(10)` creates a Purse with capacity 10 coins.
int count( )	returns the *number* of coins in the Purse (not the capacity)
int getBalance(String currency)	returns the *value* of all the coins in the Purse that have the given currency (ignoring case) Return 0 if no coins match the currency.
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 is OK, **`false`** if the Purse is full or the Coin is not valid (value <= 0).
Coin[ ] withdraw(double amount, String currency)	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 how much money is in the Purse

UML Diagram



Example: A Purse with capacity 3 coins.

```
Purse purse = new Purse( 3 );
purse.getBalance("Baht")     returns 0
purse.count( )               returns 0
purse.isFull( )              returns false
purse.insert(new Coin(5,"Baht"))     returns true
purse.insert(new Coin(10,"Baht"))    returns true
purse.insert(new Coin(0,"Baht"))     returns false. Don't allow coins with value <= 0.
purse.insert(new Coin(5,"Yen"))      returns true
purse.count( )               returns 3
purse.isFull( )              returns true
purse.getBalance("Baht")     returns 15
purse.toString()             returns "Purse with 15 Baht, 5 Yen"
purse.withdraw(11,"Baht")    returns null. Can't withdraw exactly 11 Baht.
purse.withdraw(15,"Baht")    returns [Coin(5,Baht), Coin(10,Baht)]
purse.getBalance("Baht")     returns 0.
```

5.2 Thoroughly test the Purse

Thoroughly test the `Purse` class. Try both valid and invalid inputs.

Hints for Withdraw

1. When you are trying to withdraw money, use a temporary list to hold the Coins you want to withdraw. Each time you add a Coin to the temporary list, deduct its value from the amount you need to withdraw. If the amount remaining is reduced to zero then you succeeded.

2. Before starting the withdraw, sort the coins.

3. Then, start from the end of the list (coin with biggest value) and work back toward the start of the list. It may be helpful to first find the start-index and end-index of list elements having the currency you want, so you don't need to check currency each time.

4. Don't use coins.removeAll(templist) because it will remove *all* coins that are equal (using equals) to any Coin in templist. Instead, use a loop and remove Coins one-by-one.

5. withdraw returns an array, so you need to copy your tempList to an array. Use ArrayList.toArray to copy the coins:

```
Coin[] array = new Coin[tempList.size()];
tempList.toArray(array);
```

5. Console User Interface

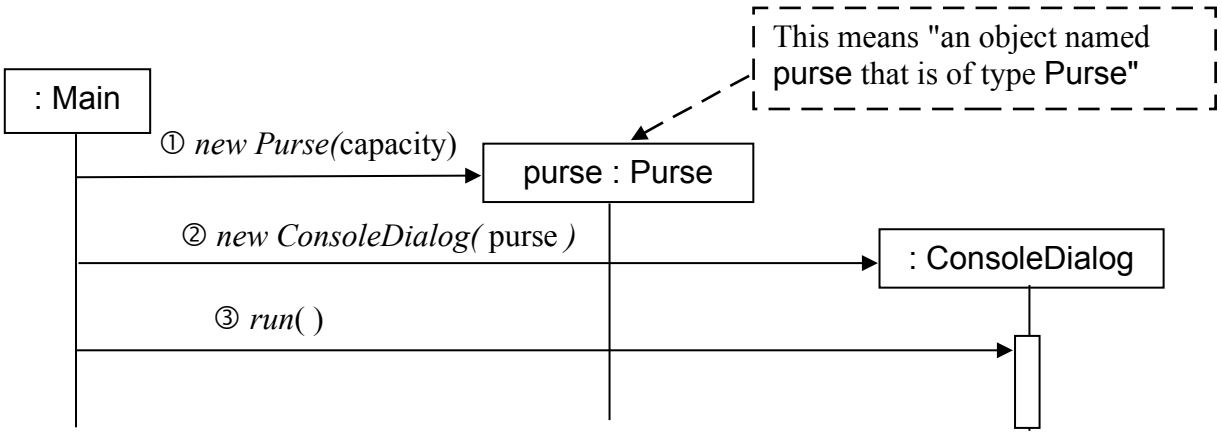
For this lab, you can use the **`ConsoleDialog`** in the sample code. No coding necessary. We will use a boring Console interface.

The ConsoleDialog needs a *reference* to the Purse so it can call the purse's methods. We want it *use a Purse*, not create one. So, we ***set*** *a Purse reference* in the ConsoleDialog constructor:

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

6. Write a Main class to create objects and start the program

Write a **`Main`** class with a static **`main`** method to create objects and "connect" them together and start the program. The **`main`** method implements 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 class creates objects and starts the application.
 */
public class Main {

   public static void main( String [] args ) {

   //TODO create the purse
   //TODO create ConsoleDialog and give it the purse
   //TODO run ConsoleDialog
   }
}
```

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;` `coins = new ArrayList<Coin>( );`
Number of items in a list	`int size = coins.size(); // like array.length`
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 c = coins.get(0);`
	`coins.remove(c);`

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 );`
	`// Another way. 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>( );`
	`list.add( ... );  // add some elements`
	`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 list2 into list1`