Instructions	Implement a RecursiveWithdraw strategy for the Purse that uses recursion to decide what items to withdraw. Write JUnit tests that include cases where (a) greedy withdraw fails but withdraw is possible (recursion should succeed), (b) edge cases to verify recursion is correct, including some case where withdraw should fail.
What to Submit	Add the code to your coinpurse project and push it to Github.

Withdraw Strategies

The original withdraw algorithm for the Purse tries to remove the highest value items first, then add smaller items until it gets the requested amount. This is called the *greedy algorithm*.

Unfortunately, it doesn't always work. Suppose the Purse contains these coins:

5-Baht, 2-Baht, 2-Baht, 2-Baht

if we try to withdraw 6-Baht, the greedy withdraw algorithm fails. There are *many, many* cases like this, so don't try to fix greedy withdraw with some cludgy code like "if it fails then I'll ignore the first item and try again".

For this application, a strategy that will *always* find a solution (if it exists) is recursion.

Problem 1: Write JUnit Tests for Failed Withdraws

1.1 Add some additional tests to the PurseTest class to test cases where a withdraw should be possible, but the greedy withdraw fails.

1.2 Also add at least one test for some edge cases where recursion might make a mistake, such as a deep recursion, e.g. alternate lots of 100-Baht and 1-Baht items in purse and try to with 80-Baht (needs to recurse many levels)

Problem 2: Implement a Recursive Withdraw Strategy

Write a RecursiveWithdraw strategy to compute a withdraw using *recursion*. The algorithm is similar to the **`groupSum`** problem on **`codingbat.com`**.

2.1 Write a RecursiveWithdraw strategy class in the coinpurse.strategy package.

2.2 Implement the recursive withdraw using test-driven development. This means to code a little, then run the unit tests, then code some more, test again, etc. until the code is complete.

2.3 When your code passes all tests, take a break and then perform *Code Review*. Read your RecursiveWithdraw code line-by-line and explain to yourself what each line does. Try to find bugs that testing missed. If you find any "missed" bugs, create a new JUnit test that does detect it.

Programming Hints

1. At each step of recursion, select *the first item* (or last item) in the money List and consider 2 cases:

Case 1: Choose this item for withdraw. By recursion, try to withdraw the *remaining* amount = amount - value of this item, using only the remaining items in the list.

Case 2: *Don't* use this item for withdraw. By recursion, try to withdraw the *entire* amount using only the remaining items in the list.

2. For the recursive step, create a *sublist* of the current list that excludes element 0.

For example:

```
withdraw( amount, currency, money):

// select the first item in the list, and recursively try 2 cases
Valuable first = money.get(0);
// Case 1: select this item for withdraw. The currency must match! (test it)
// and try to withdraw the remaining amount using the rest of the list.
// Some code is missing (you don't want it to be too easy, do you?)
remaining = amount - first.getValue();
List<Valuable> result = withdraw( remaining, currency,
                                  money.subList(1,money.size()) );
// Did the recursive withdraw succeed?
// If yes, append (or prepend) first to the result.

// Case 2: don't use this item for withdraw.
// If Case 1 didn't succeed or currency of first item didn't match,
// try to withdraw the entire amount using the other items in the list.
result = withdraw( amount, currency, money.subList(1,money.size()));
```

`list.subList(start, end)` creates a *view* of the list starting at index `start`, and up to (but not including) index `end`. It is a *view,* not a copy. This is efficient -- no copying of the List. If the List is modified, the view will change, too.

3. During recursion, at some point it will either fail or succeed. If it succeeds, create a new list for the return result and add whatever element you want to return. Higher level callers will *append* their item on this return list -- **don't create a new list at each level**!

Does Recursion Eventually Stop?

You should ensure that recursion eventually stops.

How to See What RecursiveWithdraw is doing?

The withdraw strategy object doesn't print anything, so it is hard to debug.

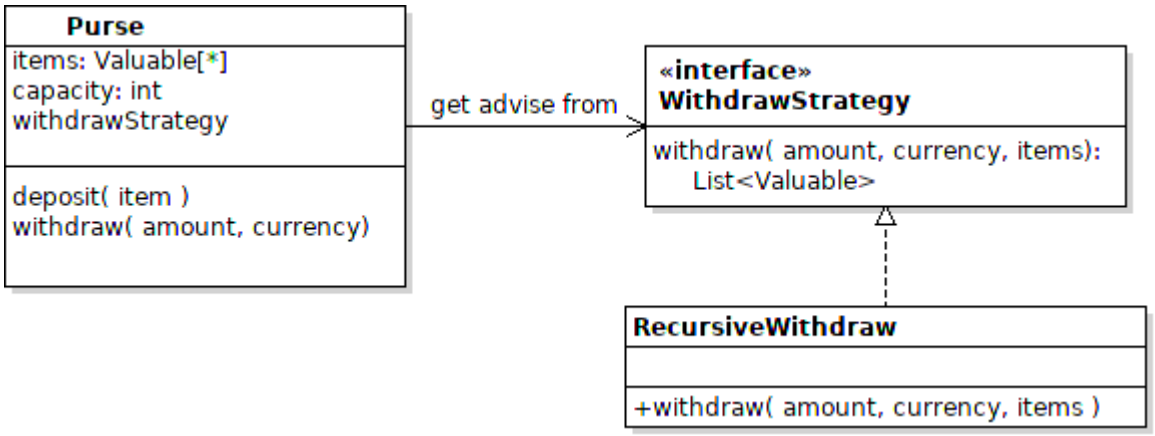
You could add code at the beginning and end (before every return) to print something. To keep you code clean, write methods named enter() and leave() to print whatever is useful to you. For example:

```
public List<Valuable> withdraw( ... ) {
    enter(amount, money);
    // do something
    List<Valuable> result = ...
    // call leave before returning.  leave() will return its own parameter
    return leave(result);
}
//TODO: write enter() and leave() to print a message on console.
// The leave method should return result, too.
```

Reference

Big Java chapter 13 covers Recursion.

codingbat.com "Recursion-2" problem set covers recursion with backtracking. The `groupSum` problem is exactly like this one.


Purse
items: Valuable[*]
capacity: int
withdrawStrategy
deposit(item)
withdraw(amount, currency)
get advise from
«interface»
WithdrawStrategy
withdraw(amount, currency, items):
List<Valuable>
RecursiveWithdraw
+withdraw(amount, currency, items)

Example of Recursive Withdraw: (diagram formatting needs correction)

Here is an example recursive withdraw using a list of numbers.

withdraw(amount, list) - try to withdraw amount from list of Number. Returns a List of elements to withdraw.

Example: Recursive withdraw for amount = 4, list = {1, 2, 2, 5},

withdraw(4, {1, 2, 2, 5})

return {2, 2}

Select the first element from list (1) and consider two cases

withdraw(4, list)

case 1: use the element (1) as part of withdraw solution. Try to withdraw remaining 3.

null (**fail**)

withdraw(3, {2, 2, 5})

case 2: don't use the first element. Try to withdraw 4 using other items.

return { 2, 2 }

withdraw(4, {2, 2, 5})

case 1: use first element (2) and try to withdraw 1 more

null (**fail**)

withdraw(1, {2, 5})

fails because the items in list are all larger than amount wanted (1).

case 1: use 2

return { 2 }

withdraw(2, {2, 5})

At this step, withdraw succeeds!
It needs to withdraw amount = 2, and the first element is the list is 2.
*So, it creates a **new list** for the solution, adds 2 (the solution) to the list, and returns the list*

the recursive step succeeded, so append another item to the solution and return it. Don't create a new list!