Hello!

Hi Everyone!

In this video I am going to go over the solution for the Flower exercise from the ruby fundamentals chapter.

In this video I'll focus on writing the two required methods for the Flower class: **sell** and **restock**

Summary of what's written

```ruby
class Flower
  # reader
  attr_accessor :name, :size, :color, :quantity_available, :bundles, :max_stock, :total_sold

  # initialize method
  def initialize(flower_dict)
    @name = flower_dict[:name]
    @size = flower_dict[:size]
    @color = flower_dict[:color]
    @quantity_available = flower_dict[:quantity_available]
    @bundles = flower_dict[:bundles]
    @max_stock = flower_dict[:max_stock]
    @total_sold = flower_dict[:total_sold]
  end
end
```

I'll get started by explaining **initialize** method and **attribute accessor** method that I have already written.

These methods both have a special meaning to ruby. When we create an instance of the Flower class, **initialize** is called with the arguments that we pass to new. In this way we can initialize a new Flower with particular attributes – or instance variables.

attr_accessor is also a special ruby method that allows you to both read – or get – and write – or set – the instance variables that you specify.

Now that we understand what's going on in the code, let's run a few tests to be sure that we can read and write all the attributes of an instance of the Flower class.

We'll do that with this bit of code.

```
rose = Flower.new({
  name: "rose",
  size: "medium",
  color: "red",
  quantity_available: 144,
  bundles: 24,
  max_stock: 350,
  total_sold: 15042
  })

puts "-----reader tests-----"
puts "name: #{rose.name}"
puts "size: #{rose.size}"
puts "color: #{rose.color}"
puts "quantity_available: #{rose.quantity_available}"
puts "bundles: #{rose.bundles}"
puts "max_stock: #{rose.max_stock}"
puts "total_sold: #{rose.total_sold}"
```

First we instantiate a new Flower – a rose, with the 7 attributes from the table.

Then we print all the attributes.

Let's go to the terminal and run our script.

Great, we instantiated a Flower and read all the attributes. Now we can move on to writing the sell method.

---

sell method

By refering back to the exercise requirements and the example tests, we know that our sell method should take one argument – the number of flowers you are selling – and update the **quantity_available** and the **total_sold.** If the number of flowers being sold, let's call this n, is less than or equal to the quantity available, this is straight forward algorithm – n is added to the **total_sold** and subtracted from the **quantity_available**

However, if n is greater than the **quantity__available** there is a decision to be made. We can't have negative flowers, so if we have 150 flowers available, we can't sell anymore than 150. lConsider how you'd like to handle this situation with code.

PAUSE

There are two main options.

You could tell the user that there are not **n** flowers available with a print statement, and sell no flowers

or

You could tell the user that there are not **n** flowers available and sell all the remaining flowers so that quantity_available is equation to 0.

I'll choose to program the second option. Let's write some pseudocode to outline our algorithm

```
# sell method
# if n <= quantity_available, updated total_sold and quantity_available by adding and subtracti
# if n > quantity_available, tell the user, update total_sold by adding quantity_available and
```

Now let's code this.

```
def sell(num)
    if num <= @quantity_available
      #@quantity_available = @quantity_available - num
      @quantity_available -= num
      @total_sold += num
    else
      puts "You don't have #{num} flowers. You only have #{@quantity_available}. We will sell t
      @total_sold += @quantity_available
      @quantity_available = 0
    end
end
```

…

You may have written

```
@quantity_available = @quantity_available - 1
```

or this may look unfarmiliar to you.

Let's rewrite this with the `-=` operator.

```
@quantity_available -= 1
```

…

Great. Now let's test this code. First we will test the case where n <= quantity*available. And then we will test the case where n < quantity*available. We will print the quantity*available and the total*sold before and after each call to **sell.**

```
puts "-----sell method tests-----"
puts "quantity_available: #{rose.quantity_available}"
puts "total_sold: #{rose.total_sold}"
puts "Sell 5"
rose.sell(5)
puts "quantity_available: #{rose.quantity_available}"
puts "total_sold: #{rose.total_sold}"
puts "Try to sell 200"
rose.sell(200)
puts "quantity_available: #{rose.quantity_available}"
puts "total_sold: #{rose.total_sold}"
```

Looks good.

---

restock method

Now for the restock method.

Let's remind ourself what the restock method should do.

Reorder as much as possible before the quantity hits the max stock. Flowers can only be ordered in the quantities of their bundles.

Does the restock method take any arguments?

PAUSE

no.

Let's look at the example and rewrite the values as variables. We will use this pseudocode to outl

```
If there are only 15 daisy's in stock, we want to restock to 300 as
closely as possible, without going over.
So, we need to order no more than 285 (300 - 15), but can only order in bundles of 50.
  How many times does 50 go into 285 (285 / 50)? 5!
  So our program needs to order 5 bundles of daisy's, which is 250 (5 x 50).
  That will update the quantity available to 265.ine our method.
```

```
# find the difference between the max_stock and quantity_available
# integer divide this difference by the number in a bundle (@bundles) to # get number_of_bundl
# multiply number_of_bundles by bundles to get the number of flowers_to_order
# add flowers_to_order to @quantity available
```

Remember you only want to restock if the difference between max*stock and quantity*available is greater than or equal to the number of flowers in a bundle. Thus, after you calculate this difference, you will you a conditional statement to check that this condition is met.

Ok, let's turn our pseudo code into code.

```
def restock
    # find difference between quantity_available and max_stock
    difference = @max_stock - @quantity_available
    if difference >= @bundles
    # determine the number of bundles
      num_bundles = difference / @bundles
    # determine the number of flowers
      num_flowers = @bundles * num_bundles
      @quantity_available += num_flowers
    else
      puts "You have too many flowers to restock at this time"
    end
  end
```

Now let's test our method.

First we'll print the **max_stock** and **quantity_available** for reference. We'll restock 2 times. The first time we should be able to restock because in our previous test we were left with 0 quantity_available –

With the second call to restock, we should not be able to restock. We can use a loop to call restock twice and include a couple of the same print statements.

```
puts "-----restock method tests-----"
puts "max_stock: #{rose.max_stock}"
puts "quantity_available: #{rose.quantity_available}"
2 times.do
    puts "call restock"
    rose.restock
    puts "quantity_available: #{rose.quantity_available}"
```

Looks good

Enhancements

That's it for all the required components.

Now you could consider adding enhancements.

A couple enhancements that come to my for me are a method that prints a description of the flower including it's **name, color, and size** –

and a method that tells you the current important sale information including **@quantity_available** and **@total_sold**

What do you think?

I hope you found that helpful.

I'll see you next time!