Object Oriented Programming the Ruby Way

Object oriented programming is a programming paradigm where we create entities called “objects” that have attributes, or data, and associated procedures, called “methods.” These objects interact with each other to form an application or a game. This results in more understandable and maintainable code if done correctly. Up until now, we’ve only talked about *procedural* programming.

In Ruby, everything is an object. So what does that mean? It means that every data type we work with, whether we’re talking about `Fixnum` or `String` has its own attributes and methods. Everything in Ruby can be sent messages that can tell the object to do stuff.

Objects At A Glance

Objects are a way for us to define code that links state of data and behaviors associated with that data. For example, we may need to represent a bank account. That bank account has a balance, and we can make withdrawals and deposits. A withdrawal will reduce the balance, and a deposit will increase it.

There’s a lot of logic involved with that process. For example, we can’t take out any more money than we have in the account. Instead of using local variables and a bunch of functions, we can group our logic into objects.

Objects have attributes (sometimes called properties) and methods. An attribute is something that the object is or has, and a method is something that the object can do, or that we can do with the object.

We can create a `BankAccount` object. That object can contain the balance of the account. We can then send this object messages like “withdraw 10” or “deposit 20”. These methods alter the `balance` attribute of the bank account object.

It’s common for an object’s methods to alter the attributes of the object. This is part of what we call encapsulation and abstraction. We hide how we handle withdrawals and deposits behind a simple interface.

- Encapsulation is how we hide the complexity. We “encapsulate” the logic in the object.
- Abstraction is the external appearance, or the way we work with the object.

To use another example, think of a television. To change the volume, you don’t rip open the TV and fiddle with the electronics. You use buttons or a remote. The buttons or remote is the abstraction. The encapsulation is how those buttons work inside of the TV.

Our goal is to hide that complexity, keeping the data and the logic encapsulated behind a simple abstract interface. By using objects, we can write code that’s easier to maintain and easier to reuse.

Messages, not methods

If you’ve programmed with objects before, you might be used to “calling methods.” However, Alan Kay, the creator of the Smalltalk programming language (which Ruby, JavaScript, and Objective-C all come from“ defined object oriented program as ”objects sending messages to other objects."

In Ruby, we define the messages that our objects can respond to, and those become our methods.

Take the `String` object. If you had a `String` like `"homer"` and you wanted to know how many letters were in that word, you'd do this:

```
"Homer".send :length
```

This 'sends a message" to the object, saying "Tell me how long you are."

We can even see if "Homer" can respond to that message before we send it:

```
If "Homer".respond_to? :length
  "Homer".send :length
end
```

However, this is a little uncomfortable for some developers who are used to more traditional object oriented language constructs, and so Ruby provides a familiar interface: the dot notation.

```
"homer".length
```

Defining Objects using Classes

A Class is our "blueprint" for creating objects. We have to write some code that defines the attributes and methods our object has, so we do that with a class.

In Ruby, classes start with a capital letter and use SnakeCasing. Here's how we define our class:

```
class BankAccount

end
```

That's it! We've defined a new object, and we can now use it in our code. We use our class to create an **instance** of an object.

```
account = BankAccount.new
```

Each instance we create is unique, with its own location in memory. We could have 10 different `BankAccount` instances and each would be its own entity in our system, all created from the same class.

Now, we haven't defined any methods on this object yet, but it turns out that this object already has several because every object in Ruby "inherits" some behavior from a basic object. We don't have to code it that way; Ruby just takes care of that for us. We'll talk more about inheritance soon. But for now, let's see what happens when we call the `to_s` method. We know that every object has a `to_s` method that converts the object to a String so it can be displayed. So what does this one look like?

```
account.to_s
=> #<BankAccount:0x007fb66b04c1c0>
```

That's not very helpful, so let's define our own by defining the `to_s` method in our object. If we do that, then when we call `to_s` Ruby will use ours instead of the default.

This is called “Overriding” a method.

Methods are just functions, so we use the `def` keyword to define them. Let’s just have our new method return `$0.00` for now, just to see if it works:

```
def to_s
  "$0.00"
end
```

So now, we need to be able to add money to the account. That means we’ll need a place to store the balance, a way to view the balance, and a way to add and remove money from the account.

Working With Attributes and Methods

As mentioned previously, objects in Ruby have attributes and methods.

An **attribute** is just a variable “inside” of the object that holds data about that object, like our `balance` for the `BankAccount.`

A **method** provides a way of interacting with the object. Methods define the messages that a Ruby object can respond to.

In Ruby, methods are just functions defined inside of a class. But attributes are nothing more than a variable that everything inside the instance of an object can see. We call those **instance variables.** Nothing outside the object can “see” the instance variable or its value.

Some languages, like JavaScript, have **properties** which let a developer read and write attributes directly to an object’s instance. Other languages allow you to declare an attribute as “public”, which means it can get read and set directly. Ruby doesn’t have features like that. It only has methods. The attributes of an object must be stored in *instance variables*. Those instance variables can only be accessed through methods.

Let’s see how we do that

Defining a “getter” method

We want to be able to retrieve the current balance of our account. To do that, we need to create a method that lets us retrieve the balance.

In Ruby, we don’t use the `get` and `set` prefixes we might use in Java. Instead, because attributes can only be accessed through methods, we simply give the getter method the same name of the attribute. So a method that retrieves the current balance looks like this:

```
def balance
  @balance = 0 if @balance.nil?
  @balance
end
```

In case that code isn’t entirely clear, we’re checking to see if the balance is `nil`, which is Ruby’s way of saying “this variable has no value.”

You may see someone use this popular Ruby shortcut:

```
def balance
  @balance ||= 0
end
```

This does exactly the same thing. It means “Return the balance as is, unless its nil, in which case set the balance to 0.” And since Ruby methods always automatically return the last executed statement, the balance’s current value is returned.

So now, when we create a new instance of our `BankAccount` object we’ll be able to see the balance:

```
account = BankAccount.new
puts account.balance
=> 0
```

And hey, this would be a great time to change the `to_s` method so that it uses the `balance` method we just created instead of a hard-coded value:

```
def to_s
  self.balance.to_s
end
```

We use `self` to tell Ruby that we’re looking for a method on the object. It’s like `this` in JavaScript or Java. If we leave it off, it’ll try to find a local variable with that name. And that can be bad if we’re trying to set a value.

Ruby developers tend to prefix methods that deal with attributes with the `self` keyword to avoid confusion.

Now, of course, we’ve lost the dollar sign; it’d be nice if this looked like money when printed out. We can fix that with the handy `sprintf` function in Ruby which takes a formatting pattern and a value:

```
def to_s
  sprintf("$%.2f", self.balance)
end
```

You can read up on `sprintf` at http://apidock.com/ruby/Kernel/sprintf

So now, when we print out the object, we'll get a nicely formatted bit of currency:

```
account = BankAccount.new
puts account.to_s
=> $0.00
```

And actually, we don't have to call `to_s` - we can just print out the object itself, which automatically calls the `to_s` method:

```
puts account
=> $0.00
```

You Try It

Create the file `bank_account.rb` and implement the object up to this point.

If you've already done this, you should do it again without looking at your notes.

You need

- A class called `BankAccount`
- A `balance` getter method that initializes the instance variable to 0 if it's not already set

- A `to_s` method that overrides the default `to_s` method and prints out the code in money.

Adding Money To The Account

Now we need to define a method to deposit money into the account. To alter the balance, we want to send the object a message that says “deposit x amount of money.” The method associated with that message should then add the amount we specify to the balance.

Can you figure out how to declare it? Well, here it is if you need help.

```
def deposit(amount)
  @balance += amount
end
```

Notice that the argument `amount` doesn’t have a type specified. We’re going to assume it’s a number.

Let’s test it out:

```
account = BankAccount.new
puts account.balance
=> 0

account.deposit 20
puts account.balance
=> 20
```

Awesome! It worked. Let’s try adding a couple more deposits:

```
account.deposit 20
account.deposit 40.0
puts account.balance
=> 80.0
```

Looks great! But there's a bit of a problem. What happens if we create a new account instance and try to put money into the account?

```
account2 = BankAccount.new
account2.deposit 20
=> undefined method `+' for nil:NilClass (NoMethodError)
```

The program fails to run because the value of `@balance` is nil **until we call the `balance` method that retrieves it.** If you look back at our first try, I displayed the balance **before** I attempted a deposit.

What we probably want to do is give the `@balance` instance variable a default value right away. The problem is that Ruby only lets us assign default values to instance variables in methods.

So we solve this with an initializer.

Defining an Initializer

An initializer lets us define code that gets called when we create a new instance of an object. It's almost like a constructor in other languages.

Let's create an initializer for our object that sets the balance to `0`:

```
  def init
    @balance = 0
  end
```

That’s all there is to it.

Now let’s remove the logic in the `balance` method that checks for a `nil` value:

```
def balance
  @balance
end
```

Much cleaner.

You Try It

Add the `init` method and the `deposit` method to the `BankAccount` class. If you’ve been following along, you should do it again for practice.

Then define a method called `withdraw` that removes money from the account. Don’t worry about withdrawals that are larger than the balance. We’ll handle that later.

Creating a Setter Method

We’ve created a getter method for the `balance`, but how do we create a setter so we can change the value of an attribute? Well, by using a method, just like what we did with the deposit. Remember, in Ruby, there are no properties, like other languages. We only have instance variables and methods.

However, it’s still convention that, if we want to create a true getter method, we make the method have the same name as the attribute's name we're retrieving. And the setter's name follows a similar pattern.

To illustrate this, let's use a differferent attribute. We don't want to create a public setter for the balance; we only want to change the balance through withdrawals and deposits. That way we might be able to log them easier, or add other logic in. **Contrary to what many books and tutorials lead you to believe, not every attribute needs getter and setter methods!** Good object-oriented design demands that you encapsulate your data and only expose what you must.

Accounts might get a name though, so let's create a way to set an account name.

```
  def name=(new_name)
    @name = new_name
  end
```

By convention, a setter method in Ruby uses the name of the attribute, followed by an equals sign. In the body of the method, we simply set the value to the argument we take in.

To test this out, let's make a getter method for this:

```
def name
  @name
end
```

Now we can set the name and then retrieve it:

```
account = BankAccount.new
account.name = "TestAccount"
puts account.name
```

Notice this line?

```
account.name = "TestAccount"
```

Ruby lets us put a space in front of the `=` sign so that it's more readable! **It actually looks like we're assigning a value to the attribute!

You Try It

Create a method called `summary` that returns the following string:

`The account "TestAccount" has a balance of $40.00.`

Of course, you shouldn't hard code the name and balance in your output. You should use the values of the object's attributes.

Test your new string like this:

```
account = BankAccount.new
account.deposit 20
account.deposit 20
account.name = "TestAccount
puts account.summary
```

You Try It

Create a method called `history` that prints out a history of withdrawals and deposits and then displays the summary. This will require that you record the history of each deposit and withdrawal. What data structure might you use to record that history?

Test your code like this:

```
account = BankAccount.new
account.name = "TestAccount
account.deposit 20
account.deposit 20
account.withdraw 30
account.deposit 100
puts account.history
```

The output should be

```
History for account "TestAccount:"
Deposit of $20.00
Deposit of $20.00
Withdrawal of $30.00
Deposit of $100

The account "TestAccount" has a balance of $110.00.
```

Simplifying Our Object With Accessor Methods

Our object is working great, but by using some idomatic Ruby, we can reduce the code we have in our object.

The pattern of a getter and setter for an attribute is so common that Ruby has a declarative syntax for its objects that we can use.

So instead of writing code like this:

```
def name
  @name
end

def name=(new_name)
  @name = new_name
end
```

We can simply add this right after the class declaration:

```
attr_accessor :name
```

The `attr_accessor` keyword defines a getter and setter method for the given attribute runtime.

But if we only wanted a getter, we could just use `attr_reader`. And if we only wanted a setter, we can use `attr_writer`.

Now, our class looks like this:

```
class BankAccount
  attr_reader :balance
  attr_accessor :name

```

we can remove the `balance` method and the `name` and `name=` methods.

Inheritance and Composition

The concept of inheritance is really quite simple; we can create objects from scratch, or we can create objects that inherit attributes and methods from existing objects. The concept is simple, but it quickly runs out of control if we don’t think carefully about what it is we are doing.

In object oriented languages, the **object is the data type**. Let’s use the classic example: animals.

```
class Animal
  def make_sound
    ""
  end

  def sleep
    "zzz"
  end
end
```

Now, we may have a Dog that “inherits” from `Animal`.

```
class Dog < Animal
  def make_sound
    "Woof!"
  end
end
```

When the dog speaks, it says “Woof”. We **overrode** the `make_sound` method. However, the `Dog` inherits the `sleep` method from `Animal`:

```
dog = Dog.new
dog.make_sound
=> "Woof!"
dog.sleep
=> "zzz"
```

This lets us represent hierarchies and easily share behavior. We don't have to repeat ourselves. But there's a huge drawback that languages like Ruby recognize, and it's really important:

The dog is no longer an animal.

It's a dog.

See?

```
dog = Dog.new
dog.class
=> Dog
```

It's not an `Animal`. It's a subclass of `Animal`, but we can't really tell that at face value. We've gone and changed its data type because we want it to have different behaviors.

The behaviors, though, are all we really care about. So why change the type?

And more importantly, very few things in web applications really have a hierarchy like the typical "Animal" example you see.

Let's look at another example to illustrate this point: a Person:

```
class Person
  attr_accessor :first_name, :last_name, :age

  def full_name
    "#{first_name} #{last_name}"
  end
end
```

So now what if we needed an administrator? We could inherit from `Person`:

```
class Admin < Person
  def is_admin?
        true
  end

  # other admin-only methods
end
```

That seems to make a lot of sense. But what if we needed a customer?

```
class Customer < Person
  def is_customer?
    true
  end

  # other customer type behaviors
end
```

Still works. But now let me throw a wrench into our plans - what if we wanted a user that could be a customer and an admin? That’s not out of the ordinary.

```ruby
class CustomerAdmin < Admin

end
```

Well, now we have the beginnings of a complex type hierarchy, and that is going to be very difficult to maintain in the long run. Now let me ask this question:

Is a Customer a Person? How about an Admin?

Ruby developers believe that we shouldn't care about what something is. An object's data type shouldn't necessarily determine its behavior. Ruby developers believe we should care about what messages the object understands. We call this `Duck typing` - if it walks like a duck and talks like a duck, it's a duck. It doesn't matter if it IS a duck.

Coupling behaviors too tightly to something's data type can lead to problems. You'll see this all over Ruby's codebase. The `Array` and `Hash` objects both have the `map` and `reject` methods, but they don't get those methods from a parent class; they get them from something called `Enumerable`, which is something called a *module*.

Using Modules for composition

A Module is like a class, except that we can't create instances of it. A module simply holds methods. That module can then be "mixed in" to an existing object, either at design time. In other words, behavior that must be shared across objects goes in a module.

Instead of inheritance, Ruby developers compose what they need with modules.

Here's how it works. Let's say we needed the ability to log messages to a text file, and we wanted this `log` method to be available in a few objects in our system.

First, we create a module called `Logger`:

```ruby
module Logger

end
```

Then, inside that module we create our `log` method that takes in a message and prints a line of text into our logfile:

```ruby
  def log(message)
    output =  "#{Time.now} : #{message}"
    File.open("log.txt", "w") do |logfile|
      logfile << output
    end
  end
```

The `File.open` method handles the opening and closing of the file by using a Ruby block. We use the `<<` operator to “push” our `output` string into the file.

With this in place, we can add it to our `BankAccount` class:

```ruby
class BankAccount
  include Logger

  # the rest of your code
end
```

The `include` keyword adds all of the methods in a module to our object as *instance methods*. We didn’t have to to use inheritance for this, and we didn’t have to inject the logger in another way. All instances of the BankAccount will have a `log` method.

Now we can use the `log` method in our object.

Best of all, we can change how the logging works without changing any classes.

You try it

Create a module called `Interest` that includes a method called `calculateInterest` that calculates interest o on the account using a simple interest formula. Modify your `BankAccount` object so it includes this new module.

Modifying object instances

Remember our `Admin` and `Customer` problem? Well, we can actually apply Ruby modules to object **instances**. The `extend` method on an object lets us do just that. First, we need to declare our behaviors for `Customer` and `Admin` as modules.

```
module Admin
  def is_admin?
    true
  end
end

module Customer
  def is_customer?
    true
  end
end
```

Then we define our class:

```
class Person
  attr_accessor :first_name, :last_name, :age

  def full_name
    "#{first_name} #{last_name}"
  end
end
```

And now we can create an instance of our class and add the modules at runtime:

```
customer_admin = Person.new
customer_admin.extend Admin
customer_admin.extend Customer
```

By using Ruby's `respond_to?` method, we can then test to see if our new object can respond to the messages we want to send:

```
customer_admin.respond_to? :is_admin?
=> true
```

feature is mostly used when building things for other developers. It's incredibly flexibile but can be very hard to figure out

Exceptions

Things go wrong occasionally. We need a way to handle that. We can handle things gracefully and defensively, or we can raise an **exception** which, if not handled by another part of our program, will cause the program to halt.

To illustrate, let’s alter the `deposit` function and ensure that what we’ve passed into the `deposit` function is a number, not a String.

All numbers in Ruby inherit from `Numeric` and so we can check the type of the object using the `is_a?` method. So, we can raise an exception like this:

```
def deposit(amount)

  if amount.is_a? Numeric
    @balance += amount
  else
    raise ArgumentError, "Deposit must be a numeric value" unless amount.is_a?
  end
end
```

If the argument passed in is a `Fixnum` or a `Float` or anything that comes from `Numeric`, we’re in great shape and we can add to the balance.

The new balance still becomes our method’s return value because *it is the last executed statement.*

However, I could rewrite that to this:

```
def deposit(amount)
  raise ArgumentError, "Deposit must be a numeric value" unless amount.is_a? Numeric
  @balance += amount
end
```

I’m using a **guard clause** here. If there’s an exception to be raised, the rest of the method won’t finish!

However, throwing an exception is only half of the battle. We need to catch the exception we've thrown. We do that like this:

```
begin
  deposit "20"
rescue Exception => e
  puts "it broke"
end
```

The `begin` keyword defines a block of code that we're going to "try" to do. The `rescue` part is where we handle the exception that was raised.

Now, unfortunately, this is very vague. This will catch *any exception* raised, not just the `ArgumentError` we specified. We can be more specific:

```
begin
  deposit "20"
rescue ArgumentError => e
  puts "You need to enter only numbers."
rescue Exception => e
  puts "It broke some other way."
end
```

Of course, `ArgumentError` is pretty generic itself. So let's make our own Exception!

Defining An Exception

An exception in Ruby is just another object that inherits from `StandardError` or `Exception`. Typically you should make your custom exceptions inherit `StandardError`, as `Exception` covers errors with the Ruby environment too.

So, here's our exception:

```
class WrongDataTypeForDepositError < StandardError
end
```

That's it! Now we can use it in our code:

```
def deposit(amount)
  raise WrongDataTypeForDepositError, "The deposit amount must be a number." unless am
  @balance += amount
end
```

Now, when we use this:

```
deposit "20"
```

we see:

```
WrongDataTypeForDepositError:
```

We can, however, embed that message right in the exception itself. All we have to do is define a `to_s` method!

```
class WrongDataTypeForDepositError < StandardError
  def to_s
    "The deposit amount must be a number."
  end

end
```

And then we can take the exception message out of our `deposit` method:

```
def deposit(amount)
  raise WrongDataTypeForDepositError unless amount.is_a? Numeric
  @balance += amount
end
```

This is much, much cleaner.

An exception class has an initializer that takes in the object and the operation so you could write something that logged the exception somewhere. Just define that method, assign the object and operation to attributes, and then you can pass the exception to a logger that can work with it.

You Try It

Create the custom `WrongDataTypeForDepositError` exception and apply it to the `deposit` method.

Then create a similar exception for withdrawals and apply it to the `withdraw` method.

Then modify the `withdraw` account so that it raises its own error if the amount of the withdrawal is more than the amount in the balance. Use a custom exception.

Class Methods (Or, Classes are Objects Too!)

A class method is a method we call on the class itself. We don't need to create an instance of the class first. This is what a Java developer might call a `static` method.

The `new` method we've used to create objects is a class method:

```
account = BankAccount.new
```

To declare a class method, we prefix the method name with `self.`

```
class Demo
  def self.foo
    "This is a class method
  end

  def foo
    "This is an instance method"
  end
end
```

Wait a minute! Didn't we say that `self` referred to "the object"? It certainly does. But it all depends on what context you're in.

- Inside of an instance method, `self` refers to the object's instance.
- Ouside of an instance method, `self` refers to the object itself.

In Ruby, classes *are* objects. Classes have their own set of predefined methods, and we can add our own. And this is the perfect place for us to add our own initializer.

If you come from Java, you might have heard about "constructor overloading" which is where you define several constructors for your object that take in different arguments. You may want to create a `BankAccount` with an initial balance you specify, or an initial name, or with the balance and the name.

In Ruby, there's no constructor overloading. We can achieve the same thing with class methods. We'll create a class method called `new_with_name` that takes in the name of the account as its argument. This method will create a new instance of an account, assign the name, and return our new instance:

```
class BankAccount
  def self.new_with_name(starting_name)
    account = BankAccount.new
    account.name = starting_name
    account
  end

end
```

Try it:

```
account = BankAccount.new_with_name("Homer's Savings")
account.name
=> "Homer's Savings"
```

Now, what about the initial balance? Can we make that work? We never made a public setter for the `balance` so we can't do it, can we?

Well, actually, we can.

```
  def self.new_with_starting_balance(starting_balance)
    account = BankAccount.new
    account.instance_variable_set :@balance, starting_balance
    account
  end
```

Ruby provides the `instance_variable_set` method to alter the instance variables of an object. In other words, we just bypassed any protection. If you know other languages, this may frighten you a bit. You should know that **this is to help you solve problems like this**, not to be abused. Just because the language allows this doesn't mean you should do it.

Really, what we should do is reconfigure our default initializer so that it takes several arguments with default values:

```
def initialize(starting_balance = 0, starting_name = 0)
  @balance = starting_balance
  @name = starting_name
end
```

When we define the arguments, we use the `=` sign to assign initial, default values to the arguments. If we don't supply the argument, the argument gets the default value.

Observe:

```
account = Account.new
account.balance
=> 0
account.name
=> nil
```

We create a new account and pass no arguments. The balanceis 0 and the name is nil.

Let’s pass a balance:

```
account = Account.new(25)
account.balance
=> 25
account.name
=> nil
```

We left off the name. We’re allowed to do that. We can leave off all optional arguments, or we can leave off some. But we have to go from left to right. We can’t omit the first argument because then things will get shifted. So we have to specify it:

```
account = Account.new(0, "Homer's Savings")
account.balance
=> 0
account.name
=> "Homer's Savings"
```

Here we’ve given a default value to the balance and specified the name.

You Try It

Alter the two class methods that we use as initializers to use the new initializer.

Public and Private and Ruby

In Ruby, all methods are public by default. That means that they are “exposed” to the outside world and can respond to those messages.

Our `BankAccount` object can respond to the messages `deposit`, `withdraw` and `balance`, as well as `name` and `name=`.

We also added a module called `Logger` to our system that added a nice `log` method to our object. That `log` function is pretty handy if we wanted to log transactions from inside our `deposit` or `withdraw` methods, but we probably don't want other parts of the program using it.

We can make methods `private,` which means that only the methods of the object (and classes that inherit from that object) can call those methods.

To make a method private in Ruby, you simply put the keyword

```
private
```

above the methods you want to be private. Typically, you place all public methods at the top of the class and all private methods at the bottom.

You Try It

In your `Logger` module, place the `private` keyword above the `log` method. Then create a new instance of your `BankAccount`. You should be unable to call the `log` method from that instance without getting an error.

Private isn't private

There are no real private methods in Ruby. They are a contract only, telling other developers that you do not wish for them to call those methods.

Organizing Code

In Ruby, we’re not required to put modules and classes in their own files, but we typically do on larger applications.

A class’s name should be lower-case, and should use underscores to separate words, so our `BankAccount` class will go in a file called `bank_account.rb`.

We’re not required to put each exception in its own file, so we’ll put all of our exceptions in a file called `exceptions.rb`

Our `Logger` module will go in its own file called `logger.rb`.

To keep our code organized, we’ll put all of these in a folder called `lib`

Here’s the structure we’ll end up with:

```
banking/
|
+--- lib/
|    |
|    +-- bank_account.rb
|    |
|    +-- exceptions.rb
|    |
|    +-- logger.rb
|
+-- app.rb
```

The last file there, `app.rb` is where our application logic can go. Here’s what it might look like:

```ruby
# Load the other files in
require_relative 'lib/logger'
require_relative 'lib/exceptions'
require_relative 'lib/bank_account'

# Create our account object
account = BankAccount.new

keep_going = true
while keep_going do

  # prompts
  puts %Q{Deposit or Withdrawal? ("d" or "w")}
  transaction = gets.chop

  puts "Amount?"
  amount = gets.chop

  # processing
  amount = amount.to_f

  if transaction == "d"
    account.deposit amount
  else
    account.withdraw amount
  end

  # Output
  puts "Your new balance is #{account}"

  puts %Q{More transactions? ("y" to continue)}

  # condition
  keep_going = gets.chop == "y" || gets.chop = "Y"
end
puts "Thank you!"
```

You try it

Split apart your objects into the file structure and write your own user interface for your program.

Modules For Organization.

Classes can be defined inside of modules, which lets us use modules for namespaces. For example:

```
module MyBankingApp

  class BankAccount

  end

end
```

Now we can create our object instance like this:

```
account = MyBankingApp::BankAccount
```

This way we can ensure that our code doesn't collide with the names of other objects in the system.

Wrapping Up

We covered a lot this week:

1. We learned how to encapsulate behavior and data in objects
2. We learned about message passing
3. We used instance variables, getters, and setters
4. We implemented attribute accessors, readers, and writers
5. We explored inheritance and composition
6. We used class methods

Homework

Create a simple program that displays 5 items (names and prices) to the user. Let the user choose an item to add to the cart. Once they've added an item, ask if they want to add another. If they do, repeat. If not, display a summary of their cart, plus the total.

The summary should look like this:

```
Total items:

Widget: $5.00 x 1 = $10.00
Dodad: $10.00 x 2 = $20.00

Total: $30.00
```

To do this you'll need:

1. An `Item` object. An item should have a `price` and a `name`
2. A `Cart` object, which should
   - use an internal data structure, like an array to store items.
     - This attribute should not be exposed publicly via a getter method.
   - an `add_item` method that adds an item to the cart.
     - When you add an item, check to see if it's already in the array. If it is, track

    its quantity. It might be best if your array of items is a hash which stores the item *and* its quantity.
  - The cart should have a `total` method that computes the total of all the items in the cart.
  - The cart should have a method called `items` that returns the items in the cart along with the quantity. This should be a getter method only, of course
  - Your cart's `add_item` method should raise a custom exception if something other than an `item` is passed in. Use `is_a?` to check the type.
3. The `item` and `cart` objects should not contain any `puts` or `print` or `gets` statements. Those should be outside of the classes. These classes should be able to be used in a web app *or* a console application, so keep the user interface separate from the logic.