Ruby and OOP for the Old-Time C Programmer

They say the first program in any language is the same:

```
puts "Hello, world!"
```

But this is boring. Ruby uses **`puts`** ("put string") where other languages use **`print`**, **`printf`**, **`write`**, **`writeln`**, **`println`**, and so on. So what? Who cares?

Suppose you're an old-time C programmer. Suppose you're unfamiliar with Ruby and with Object-Oriented Programming in general. Suppose you've heard a little about SmallTalk, but were turned off by the syntax and the complexity (and maybe the terminology).

In that case, you're the person I'm writing for. Let's have fun.

Objects and such

First of all, let's talk just a little about OOP. What is an “object," anyway? And should we care?

Think back to how we use structs in C. They're for packaging related variables together into a single unit -- "encapsulation," if you will.

Now imagine that besides variables, actual functions can live inside such a struct. Finally, imagine that those functions all share a scope which is basically the "inside" of the struct.

This is what we call an "object." This is 50% of what OOP is all about. The mystery is half over.

When I first started to learn OOP (after college), part of the "mystique" was the terminology. I had heard of "message passing," which sounded sort of like something esoteric; but when I started to learn, I didn't really see that term used much.

To be perfectly honest, I think that technical people sometimes enjoy making up precise technical terms for things in their world. So often I have been puzzled by a piece of jargon, and then when I learn what it means, I think: "Oh, is *that* all it means? Did we even need a word for that?" So as we go along, I may expose these a little.

The first thing that tripped me up was the term "method." Although I have been used to it for years, when I stop and think about it, it still seems like a weird usage.

A *method* is a function that lives inside an object and can access the internals of that object. That's all it is.

In Ruby, everything is an object (depending on what you mean by “everything"). So for example, any string is an object, and we can call methods on it. An object "knows" things about its state -- for example, a string knows its own length, and we can "ask" it what that is by calling the length method:

```
str = "Hello"
n = str.length    # This gives us 5 as a result
```

An object can also perform operations on itself "on request" -- for example, the method **sub** ("substitute") will perform string substitution (and please note that it returns a new string):

```
s1 = "Hello"
s2 = s1.sub("o", "fire and brimstone")
puts s1       # Unchanged - prints "Hello"
puts s2       # Prints "Hellfire and brimstone"
```

Let's back up a second and notice some things that are different from C. First of all, we don't do any explicit memory allocation (or deallocation). Memory is allocated automatically "as needed" and is "garbage collected" when objects go out of scope.

There are no pointers as such -- or should I say, almost everything is a pointer? Suppose I do this:

```
s3 = "Goodbye"
s4 = s3
puts s3          # Obviously both these will
puts s4          #   print "Goodbye"
```

Don't expect that some sort of "copy" operation is going on. A variable name is just a "label" for an object. Both these names (**s3** and **s4**) refer to the same object.

Think of these variable names as sort of like pointers. They don't need to be dereferenced, because the syntax and semantics cooperate smoothly. If you "always" dereference a pointer, why should an asterisk be needed in the syntax? Why not express this universal case by the absence of any such notation?

By contrast, note that **s1** and **s2** refer to different objects. The **sub** method creates a new object on the fly which is the "result" of the operation.

Also notice that we haven't declared any variables. Local variables just spring into existence when they are assigned. In fact, the real reason we don't declare a variable as a string or integer

or whatever... is that variables simply do not have types. A variable is purely and simply a name, a reference to an object. The objects, of course, do have types.

We illustrate this by looking at an object whose value changes. In Ruby, a method name can end in an exclamation point; usually this indicates that the method not only returns the expected value, but also changes the original object in place. (The exclamation point makes me think of Steven Wright's comment, "Factorials were someone's effort to make math look exciting.")

So **sub** and **sub!** are methods that do similar things -- except that the former returns a new object, and the latter modifies the original object. Let's clarify this:

```
s3 = "Goodbye"
s4 = s3
s5 = s3.sub("bye", " heavens!")

puts s3     # Goodbye
puts s4     # Goodbye
puts s5     # Good heavens!
```

So far, so good. There are two objects here; the new one is labeled with the name **s5**, and the first one can be referred to either by **s3** or **s4**.

Let's go a step farther. Let's use **sub!** to alter the original object.

```
s6 = s3.sub!("bye", " grief, Charlie Brown!")

puts s6      # Good grief, Charlie Brown!
puts s5      # Good heavens!

puts s3      # Good grief, Charlie Brown!
puts s4      # Good grief, Charlie Brown!
```

The values of **s5** and **s6** should be absolutely what you expected. The value of **s3** has changed because we invoked **sub!** on it. To hammer this point home, notice that when we changed **s3**, **s4** also changed. That's because **s3** and **s4** are simply different names or labels for the same object. To make it perfectly clear: Suppose Bill's house is also Fred's house. If I paint Fred's house blue, then Bill's house will also be blue.

So a string in Ruby can change its value. We say this is a *mutable* object (as opposed to some that are *immutable*). This is terminology that I find "almost unnecessary."

We'll look at "real" examples of immutable objects later. But there is a special case that is worth mentioning.

Consider the integer 25. Do we really want to store this as a "fake pointer"? After all, the pointer may take up as much memory as the integer, and we'll only waste time by following that pointer.

In short, we store integers as immediate values. This means that when we manipulate an integer, we're manipulating the value itself, not through a pointer to the value. Here's an example:

```
n = k = 5
n = n + 1
puts n          # 6
puts k          # 5
```

So immediate values behave differently from other values such as strings. We can "change the value" of a variable such as **n**, but we can't change the value of the integer itself. The value of **5** will always be **5**.

Seeing **n = n + 1** you might ask: Can't I do this in a shorter way, as in C? Yes, you can do this:

```
n += 1
```

But you can't use the increment operator **++** because it simply doesn't exist in Ruby. This is arguably a "feature"; if **++** existed for integers, it would act more like assignment, operating on the variable rather than the object. We can increment **n** from **5** to **6**, but we can't change **5** to be **6**. As I said, **5** will always be **5**, even for very large values of **5**.

And by the way: If you (are so unfortunate as to) know C++, bear in mind that in Ruby, assignment is *not* an operator! That is, it works strictly with variables, not with objects.

A Little Syntax and Terminology

OK, let's take a short interlude and talk just a little about syntax. You've noticed that variable and method names look much as you would expect in any other language. A capitalized name is (almost always) a constant. Interestingly, parentheses are optional in Ruby:

```
string.sub(this, that)
string.sub this, that    # Same statement
```

Most operators are the same as you would expect: **+**, **-**, *****, and so on. But here's something interesting. I said before that I don't hear (or use) the "message passing" terminology much at all. But there is an exception in a way. The object on which the method is invoked is called the "receiver" -- that is, it "receives" the message (which is a method and perhaps parameters).

In the above example, **`string`** is the receiver. Or really it's a reference to an object that is the receiver.

So what is an "operator" in Ruby? The answer is that, in most cases, an operator is simply a method that is called in a "prettier" way, in the infix notation that we are so used to:

```
a = 3
b = 5
c = a + b
c = a.+(b)   # Same statement!
```

So when we see an operator, we can think of the lefthand side as being the receiver. Why do we care? Because when we say the receiver "receives the message" that is essentially saying that it owns or contains the method that is called.

So... in this expression, what does **`+`** mean?

```
x + y
```

The answer is that, without context, you just don't know. If **`x`** and **`y`** are integers, this is ordinary addition. If they are strings, it's string concatenation. If they are arrays, it's array concatenation.

The (natural) convention is that the receiver determines the meaning of the message. You can pass in any type of object you wish, but obviously not everything makes sense. Look at this:

```
a = "Hello"
b = " there"
c = a + b         # "Hello there"

d = 2
e = 3
f = d + e         # 5

g = a + d         # error!
```

It's obvious that adding a string to an integer (or vice versa) is nonsense. What may not be obvious is that this is a *runtime* error, not a syntax error! Ruby is very dynamic, most things happen at runtime, and variables have no types.

So the guy on the left is king, and gets to decide the meaning of the operator that follows. (This is similar to cartoon etiquette, where the person on the left speaks first.) That's basically the only reason we care what the "receiver" is.

Arrays

Now let's take a little digression. We've seen numbers and strings, but we can't yet do any "real" coding without more complex things like arrays.

Arrays in Ruby are pretty intuitive. A literal looks like this:

```
x = [1, 2, 3, 4]
```

But as the ads used to say, "This is not your father's Oldsmobile." There are two big differences between Ruby's arrays and those in C. First of all, arrays are dynamic; just as with strings, you never allocate or deallocate memory. You let them grow and shrink as needed.

Second of all, an array is not limited to storing a single data type. Each element is in effect an object (or more precisely a reference to an object). We can mix types if we choose to:

```
array = [1, 2, "three", 4.0, "five"]
```

At first glance, this may seem crazy. Soon it will feel perfectly natural. After all, every object “knows" its own type.

Obviously an array knows its own length:

```
x.length       # 4
array.length   # 5
```

That means, of course, you need never "maintain" your own count of the number of items, along with all the bugs that can ensue.

Blocks are Insanely Fun

Now here is something very special in Ruby. This might take a day or three to really grasp, but once you do, it is life-changing. So fasten your seatbelt and return your tray table to the upright position.

A code block in Ruby is not quite like what we're used to in C. It's more of a first class citizen. It's not just a syntactic grouping of statements, but a semantic unit of a sort.

Think of it this way: A method call can have one parameter on the end of the list that is a "special" parameter. The block of code itself is like a parameter. The method invokes the code block one or more times as it needs to.

Here is the most common example. We use the iterator each to iterate over each item in an array. What do we “do" with that item? Whatever we want -- we just code the block accordingly. The simplest form of block in Ruby is just written with braces:

```
array.each {|x| puts x }    # Produces five lines of output
```

Now, what is happening here? The method **each** "knows" that the array has five elements. It iterates over those one at a time and passes every one into the specified code block. Note the "vertical bars" at the beginning of the block; this is like a parameter list. Whatever the method passes in, the block will call it (in this case) **x**, which is in effect a variable local to the block. We can then use it however we like (in this case, just printing it out).

As an aside: We didn't really have to use **each** in this simple case. Ruby's **puts** method is smart enough to treat an array "sanely":

```
puts array    # Same five lines of output
```

Also, there is a **do**/**end** block that is semantically the same. We usually use **do**/**end** for multiple lines and braces for a single line, but that is only a convention:

```
array.each do |x|
  puts x
end

# Could also use these "unconventionally"

array.each do |x| puts x end

array.each {|x|
  puts x
}
```

I won't explain right now how to write a method that can call a block. (In fact, I haven't shown you how to define a method at all yet.) But trust me when I say that the ability to attach an arbitrary block of code to a method call is one of the neatest, most useful features in the language.

Hashes

Let's look at one more feature before we do a "real" coding example. There is a data structure in Ruby that we call a *hash*, and it's incredibly useful.

Internally, a hash is truly a "hash" in the computer science sense. Items of data are mapped to other items; “keys" are mapped to "values." A key can be any object.

Internally, some sort of math is done on some byte representation of the key. As coders, we don't normally care about those details. A hash can map any key to any value, and it stores and retrieves very fast. Think of it as a generalized case of an array: The indices don't have to start at zero, they don't have to be contiguous, and they don't even have to be integers.

The syntax for a hash literal is a bunch of key-value pairs (using **=>** to associate them) in braces. (If you see the braces, you might think at first glance it's a code block. In short order, your eye will pick up the difference instantly.)

Here is a hash containing a few atomic numbers:

```
atomic_numbers = { "H" => 1, "He" => 2, "O" => 8, "Li" => 4 }
```

We can access any element in this way:

```
element = "Li"
num = atomic_numbers[element]     # returns 4
```

Remember the keys and values can be any kinds of objects. Also note there is an **each_pair** iterator:

```
atomic_numbers.each_pair do |elem, num|
  puts "Element #{elem} has atomic number #{num}"
end
```

This of course produces four lines of output. Notice that I "interpolate" variables into a string here for the first time. That's fairly intuitive. They don't have to be variables, but can be completely arbitrary expressions.

A Sample Program

Now the moment you've been waiting for. It's time for a more "real-life" problem.

I'll state the problem and explain the approach. Then I'll implement it incrementally in Ruby, explaining as I go.

Problem: You have a dictionary of words, sorted, in a file of one word per line (such as **`/usr/share/dict/words`**). Find the largest set of anagrams. (For example, one such set would be "act" and "cat" -- but that is very boring, a set of only two words that are only three letters.)

Disclaimer: This idea was basically stolen from an article in Bentley's "Programming Pearls" column, where he talked about what he called "Aha!" algorithms. Dr. Rainey Little shared it with me in 1984 or so.

You might think that it's a very hairy problem to find all the anagrams in the dictionary. The "freshman" approach would be to take word 1, scan words 2 through N, permute each word and compare, then repeat for the rest of the dictionary. This is a horrible solution.

The solution Bentley proposes is: Recognize that each word has a "signature" which is its letters sorted into alphabetical order. (STAR, RATS, and ARTS all map onto ARTS.) Compute each signature, then sort by the signatures. Then all the anagrams fall adjacent to each other in output.

Before we continue, stop and ask yourself: How would I do this in C? How many lines of code would it take? How long would I have to debug?

It's possible at times to be 8 or 10 times as productive in Ruby as in C. That's highly subjective, of course, and varies with the programmer and the problem (and the phase of the moon).

But that doesn't assume mastery of Ruby first! If you're a programmer, you can grasp the basics in an afternoon; and you could be genuinely productive in less than a week.

So: I'll use Bentley's approach, but I'll use a hash. And I've extended the problem to the "largest set of anagrams." Here goes.

First of all, let's read the list of words into an array. In the 80s, this was bad form, but in the 21st century, we don't blink at it.

```
myfile = File.new("/usr/share/dict/words")
words = myfile.readlines
```

Don't worry about what **File** is right now. Being capitalized, it's a constant of some sort. Actually, it is an object class (or simply a class) which is like a built-in type in Ruby. And **readlines** is a method that returns an array of strings.

As a mild annoyance, each word comes in with a newline character. I'll trim those off with the **chomp!** method.

```
words.each {|word| word.chomp! }
```

I want to transform each word to a signature. But how do I create a signature, anyway?

Well, I can use **split** to convert a string to an array of characters (by splitting on a "null delimiter" or empty string). Then I can use **sort** on that array, and I can use join on the sorted array to convert back to a string. Here is a code fragment:

```
letters = word.split("")  # get an array of letters
sorted = letters.sort     # sort the array
key = sorted.join         # re-join into a signature string
```

That probably is fairly clear. But in the Ruby world, many would consider it needlessly verbose. Every method call returns some kind of object, and every object can have methods of its own. So we could eliminate some lines of code (and some temp variables) and write it this way:

```
key = word.split("").sort.join
# example: "conserve" maps to "ceenorsv"
```

Now I'll create a hash that will associate each signature with an array of words:

```
hash = {}
words.each do |word|
  key = word.split("").sort.join
  hash[key] = hash[key] || []
  hash[key] << word
end
```

This requires some explanation. We start out with an empty hash and loop over the list of words. We get a key (signature) for each one.

There are two cases: Either this key is already in the hash, or it isn't. If it isn't, then **hash[key]** will return a **nil** value (basically "a value that isn't a real value"). In Ruby, the only things that "test false" are the actual value **false** and the **nil** value. (Everything else, including the integer **0**, the empty string **""**, the empty array **[]**, and so on, will all test true.)

So the "or" operator **||** will short-circuit if the first operand is true. This gives us a common idiom in Ruby, so that we can "assign this value only if it doesn't have a value already."

```
x = x || y   # if x is nil, assign y to it
x ||= y      # a shorthand way to do it
```

So when I say this:

```
hash[key] ||= []
```

I am really saying, "If this key isn't in the hash, associate it with an empty array." The next line uses an unfamiliar operator **<<** which simply appends the word onto the array. A sample entry in the hash might look like this:

```
{ "arts" => ["arts", "rats", "star", "tars"] }
```

So when this loop completes, the hash will be fully populated. I want to ask myself then: What are the sizes of these sets? And which set is largest?

I can get the list of values in the hash with the **values** method. In this case, it will give us an array of arrays.

```
arrays = hash.values
```

Let me introduce here the **map** function. This can take a few minutes to understand. Basically it just takes an array and "maps" it to another array of the same size. The attached code block is used to transform each element. Here is an unrelated example:

```
nums = [1, 2, 3, 4, 5, 6]
squares = nums.map {|x| x*x }
# squares is now: [1, 4, 9, 16, 25, 36]
```

So I can get a list of the sizes of the sub-arrays in this way:

```
sizes = arrays.map {|v| v.size }
```

And I can find the largest of these sizes with the **max** method:

```
most = sizes.max
```

Then I can find the first item in the hash with that number of anagrams. (It's possible that there could be more than one "largest set," of course, with some signatures tied for first place.)

The **find** method on a hash will evaluate the code block and return the first key-value pair for which the block evaluates to true. The key-value pair is returned as an array. In this case, the second element is a sub-array.

```
best = hash.find {|sig, anagrams| anagrams.size == most }
```

Now let's print some results. The **inspect** method will print an object such as an array in a "pretty" or human-readable format.

```
puts "No word has more than #{most-1} anagrams."

# best looks like: [ signature, [word1, word2, ..., wordN] ]

anagrams = best[1]     # Ignore element 0 (the signature)
word = anagrams.shift  # Remove first one from the list
puts "The word #{word} has #{most-1} anagrams: #{anagrams.inspect}"
```

So here is the final program:

```
myfile = File.open("/usr/share/dict/words")
words = myfile.readlines
words.each {|word| word.chomp! }

hash = {}
words.each do |word|
  key = word.split("").sort.join
  hash[key] ||= []
  hash[key] << word
end

arrays = hash.values
sizes = arrays.map {|v| v.size }
most = sizes.max
best = hash.find {|signature, anagrams| anagrams.size == most }

puts "No word has more than #{most-1} anagrams."

anagrams = best[1]      # Ignore element 0 (the signature)
word = anagrams.shift   # Remove first one from the list
puts "The word #{word} has #{most-1} anagrams: #{anagrams.inspect}"
```

On my system, this runs in less than 3 seconds, and produces this output:

```
No word has more than 8 anagrams.
The word angor has 8 anagrams: ["argon", "goran", “grano",
"groan", "nagor", "orang", "organ", "rogan"]
```

This program is barely 20 lines of code, with no "real" effort to make it short. Your assignment now is to go and write it in C. See you next Wednesday.