Jason Clinton

ESSENTIAL CODE AND COMMANDS

Ruby

PHRASEBOOK

CONTENTS AT A GLANCE

the Standard Library, uses this feature to allow inter-Ruby script communication over sockets.

Duplication

```
a = Hash.new
b = a.dup
a.object_id
  #=> -605558818
b.object_id
  #=> -605579038
```

You can use `#dup` and `#clone` to duplicate objects. The difference between the two is this: `#clone` copies the state of the object in the VM to a new memory location; `#dup` generally attempts to use the class's own `#initialize` method.

Notice that the internal state of the object can refer to other objects in the VM. If so, those references still refer to the same objects—even though they were copied. For instance, consider a `Hash` of `Hashes`:

```
a = 'foo'
b = { 'bar' => 'baz' }
  #=> {"bar"=>"baz"}
c = {a, b }
  #=> {"foo"=>{"bar"=>"baz"}}

d = c.dup
  #=> {"foo"=>{"bar"=>"baz"}}

# it didn't duplicate the nested Hash
d[a].object_id
=> -739733318
b.object_id
=> -739733318
```

This happens because we are not making a deep copy. To implement a deep copy, you can use the "Marshal copy trick":

```
a = 'foo'
b = { 'bar' => 'baz' }
  #=> {"bar"=>"baz"}
c = {a, b }
  #=> {"foo"=>{"bar"=>"baz"}}

d = Marshal.load(Marshal.dump(c))
  #=> {"foo"=>{"bar"=>"baz"}}

# this time they nested Hashes are in separate
memory
d[a].object_id
  #=> -739819088
b.object_id
  #=> -739733318
```

See the chapter "Working with Collections" for examples of recursive algorithms which can be used to perform deep copies.

Protecting an Object Instance

```
a = Hash.new
a.freeze
a['Foo'] = 'Bar' # error
```

You can freeze an object, to prevent it from being manipulated, by using the `#freeze` instance method. This is useful when you want to encourage clients of your class to `#dup` or `#clone` your object before working with it.

This code produces the following output:

```
TypeError: can't modify frozen hash
```

Note, however, that frozen objects *can* be reclaimed by the garbage collector (such as when they fall out of scope).

Garbage Collecting

```
GC.start
```

Using the previous code causes the garbage collector to be invoked manually. You might use this code when you've just finished processing a very large data set and you know that now would be a good time to flush all information that has fallen out of scope.

Note that Ruby will `block` (pause execution of scripts) while the garbage collector is being run. To prevent this from accidentally happening in the middle of some critical stream-processing code, you can turn off the GC temporarily:

```
GC.disable
```

The garbage-collection system in Ruby is mostly out-of-sight, out-of-mind. And really, that's a good thing. It does a pretty good job and generally stays out of your way. But it might be interesting to know what's going on behind-the-scenes. Let's take a look at the `ObjectSpace` module.

You can use `ObjectSpace` to iterate over every item currently in the Ruby VM:

```
ObjectSpace.each_object {|x| p x }
```

However, you might want to limit it to only objects that are instances of a certain class or module:

```
# show all open files
ObjectSpace.each_object(File) {|x| p x }
```

You might also want to know *when* an object is being garbage-collected. `ObjectSpace` provides a handy way of attaching a method that will be executed when an object is deleted:

```
begin
    a = {}
    ObjectSpace.define_finalizer( a,
        proc {puts "Deleted Hash"} )
end
```

This code produces the following output when the object is GC'd:

```
Deleted Hash
```

Using Symbols

```
method(:foobar).call()
```

What does this statement mean? Read literally, one could say, "Look up the method `foobar()` and call it." Except for the overhead of calling two additional methods, this is exactly equivalent to this:

```
foobar()
```

Of all the topics in Ruby, the topic of `Symbols` is perhaps the most difficult to grasp. (To the programmer who is coming from Lisp or Smalltalk, you can think of `Symbols` as "symbols" or "atoms," respectively.) The most important thing to remember is that a `Symbol` is a unique name, contains only its own name, and always contains its own name.

It's popular to respond to queries about symbols by saying "Symbols are just immutable strings." But this analogy really doesn't hold. Perhaps the best way to

explain a `Symbol` is to cut around the analogies and go straight to the technical issue.

As you already know, function names in Ruby must be unique in the context in which they are called; otherwise, how will Ruby know which method to call? So why not simply call a method by a `String` that the programmer can personally verify is unique? Well, Ruby enables you to do that if you want:

```
method('foobar').call()
```

Notice that this is very similar to the first code snippet in this subsection. This is primarily because it has become a `convention` to automatically convert `String` parameters to `Symbols` inside methods that expect `Symbols`.

This need for uniqueness is the real reason `Symbols` are used in Ruby (or any other language) and is motivated by the way programming languages store information about the local context. When you call the method `foo()`, Ruby checks a hash table of all the methods that you (and Ruby) have defined. Because no two hashes of any `word` are ever equal, this ensures that each unique method `name` has a unique position in this hash table. Almost all programming languages use a hash to optimize lookup of methods and variables. In Ruby's case, a `Symbol` is a precomputed hash. That is, a `Symbol` is computed into its equivalent `hash value` at `parse time`. If you were to use `Strings` to refer to internal objects, Ruby would have to compute the hash value of that `String` every time it encountered it during the execution of your program.

Anywhere the literal `:my_symbol` appears in your code, it refers *immediately* to the point in the hash table where any variable or method named `my_symbol` *must* be stored. This brings up another point: In the same

way that `1 == 1` and `256 == 256`, the symbol `:foo == :foo. :foo` always computes to the same hash value; therefore, it has exactly the same value everywhere.

This property can be used throughout Ruby to speed things up a bit. This benchmark demonstrates the overhead of having to compute the `String` hash each time that a method expecting a `Symbol` receives one.

```
require 'benchmark'

def foobar
  # pass
end

n = 500000

Benchmark.bmbm do |bench|
  bench.report('Symbol') do
    n.times { method(:foobar).call() }
  end
  bench.report('String') do
    n.times { method('foobar').call() }
  end
end
```

This code produces the following output:

```
Rehearsal ----------------------
Symbol 1.020000 0.090000 1.110000 ( 1.134434)
String 1.280000 0.080000 1.360000 ( 1.392788)
------------------ total: 2.470000sec

       user     system   total    real
Symbol 1.040000 0.070000 1.110000 ( 1.133537)
String 1.270000 0.100000 1.370000 ( 1.401934)
```

Remember, it's convention to allow programmers to also use a `String` anywhere they use a `Symbol`, so be