**is\_a?**

def prime(n)

puts "That's not an integer." unless n.is\_a? Integer

is\_prime = true

for i in 2..n-1

if n % i == 0

is\_prime = false

end

end

if is\_prime

puts "#{n} is prime!"

else

puts "#{n} is not prime."

end

end

prime(2)

prime(9)

prime(11)

prime(51)

prime(97)

**Reverse as well as normal sort order**

def alphabetize(arr, rev=false)

if rev

arr.sort! { |item1, item2| item2 <=> item1 }

else

arr.sort! { |item1, item2| item1 <=> item2 }

end

end

books = ["Heart of Darkness", "Code Complete", "The Lorax", "The Prophet", "Absalom, Absalom!"]

puts "A-Z: #{alphabetize(books)}"

puts "Z-A: #{alphabetize(books, true)}"

**To know the time taken for strings vs symbols**

require 'benchmark'

string\_AZ = Hash[("a".."z").to\_a.zip((1..26).to\_a)]

symbol\_AZ = Hash[(:a..:z).to\_a.zip((1..26).to\_a)]

string\_time = Benchmark.realtime do

100\_000.times { string\_AZ["r"] }

end

symbol\_time = Benchmark.realtime do

100\_000.times { symbol\_AZ[:r] }

end

puts "String time: #{string\_time} seconds."

puts "Symbol time: #{symbol\_time} seconds."

**Movie and ratings using case**

movies = {

Memento: 3,

Primer: 4,

Ishtar: 1

}

puts "What would you like to do?"

puts "-- Type 'add' to add a movie."

puts "-- Type 'update' to update a movie."

puts "-- Type 'display' to display all movies."

puts "-- Type 'delete' to delete a movie."

choice = gets.chomp.downcase

case choice

when 'add'

puts "What movie do you want to add?"

title = gets.chomp

if movies[title.to\_sym].nil?

puts "What's the rating? (Type a number 0 to 4.)"

rating = gets.chomp

movies[title.to\_sym] = rating.to\_i

puts "#{title} has been added with a rating of #{rating}."

else

puts "That movie already exists! Its rating is #{movies[title.to\_sym]}."

end

when 'update'

puts "What movie do you want to update?"

title = gets.chomp

if movies[title.to\_sym].nil?

puts "Movie not found!"

else

puts "What's the new rating? (Type a number 0 to 4.)"

rating = gets.chomp

movies[title.to\_sym] = rating.to\_i

puts "#{title} has been updated with new rating of #{rating}."

end

when 'display'

movies.each do |movie, rating|

puts "#{movie}: #{rating}"

end

when 'delete'

puts "What movie do you want to delete?"

title = gets.chomp

if movies[title.to\_sym].nil?

puts "Movie not found!"

else

movies.delete(title.to\_sym)

puts "#{title} has been removed."

end

else

puts "Sorry, I didn't understand you."

end

**Short-Circuit Evaluation**

Recall that we have the boolean operators **and** (&&) and **or** (||) in Ruby. The && operator only returns true when the expressions on *both* sides of the operator are true; || returns true when one *or* the other *or both* of the expressions involved are true.

Ruby does this via **short-circuit evaluation**. That means that Ruby doesn't look at both expressions unless it has to; if it sees

false && true

it stops reading as soon as it sees && because it knows false && anything *must* be false.

**To print first n prime numbers using module ‘prime’**

$VERBOSE = nil # We'll explain this at the end of the lesson.

require 'prime' # This is a module. We'll cover these soon!

def first\_n\_primes(n)

unless n.is\_a? Integer

return "n must be an integer."

end

if n <= 0

return "n must be greater than 0."

end

prime\_array = [] if prime\_array.nil?

prime = Prime.new

for num in (1..n)

prime\_array.push(prime.next)

end

return prime\_array

end

first\_n\_primes(10)

**Yield**

def yield\_name(name)

puts "In the method! Let's yield."

yield("Kim")

puts "In between the yields!"

yield(name)

puts "Block complete! Back in the method."

end

yield\_name("Eric") { |n| puts "My name is #{n}." }

# Now call the method with your name!

yield\_name("Huli") {|n| puts "My name is #{n}."}

procs? There are two main advantages:

1. Procs are full-fledged objects, so they have all the powers and abilities of objects. (Blocks do not.)
2. Unlike blocks, procs can be called over and over without rewriting them. This prevents you from having to retype the contents of your block every time you need to execute a particular bit of code.
3. **Lambdas vs. Procs**
4. If you're thinking that procs and lambdas look super similar, that's because they are! There are only two main differences.
5. First, a lambda checks the number of arguments passed to it, while a proc does not. This means that a lambda will throw an error if you pass it the wrong number of arguments, whereas a proc will ignore unexpected arguments and assign nil to any that are missing.
6. Second, when a lambda returns, it passes control back to the calling method; when a proc returns, it does so immediately, without going back to the calling method.
7. To see how this works, take a look at the code in the editor. Our first method calls a proc; the second calls a lambda.

Eg:

def batman\_ironman\_proc

victor = Proc.new { return "Batman will win!" }

victor.call

"Iron Man will win!"

end

puts batman\_ironman\_proc

def batman\_ironman\_lambda

victor = lambda { return "Batman will win!" }

victor.call

"Iron Man will win!"

end

puts batman\_ironman\_lambda

NOTE: See how the proc says Batman will win? This is because it returns immediately, without going back to the batman\_ironman\_proc method.

Our lambda, however, goes back into the method after being called, so the method returns the last code it evaluates: "Iron Man will win!"

Output:

Batman will win!

Iron Man will win!

**Quick Review**

All this talk of blocks, procs, and lambdas might have your head spinning. Let's take a minute to clarify exactly what each one is:

1. A block is just a bit of code between do..end or {}. It's not an object on its own, but it can be passed to methods like .each or .select.
2. A proc is a saved block we can use over and over.
3. A lambda is just like a proc, only it cares about the number of arguments it gets and it returns to its calling method rather than returning immediately.

**OOPS**

class Language

def initialize(name, creator)

@name = name

@creator = creator

end

def description

puts "I'm #{@name} and I was created by #{@creator}!"

end

end

ruby = Language.new("Ruby", "Yukihiro Matsumoto")

python = Language.new("Python", "Guido van Rossum")

javascript = Language.new("JavaScript", "Brendan Eich")

ruby.description

python.description

javascript.description

**GLOBAL($), INSTANCE(@) & CLASS(@@) VARIABLES**

class Computer

$manufacturer = "Mango Computer, Inc."

@@files = {hello: "Hello, world!"}

def initialize(username, password)

@username = username

@password = password

end

def current\_user

@username

end

def self.display\_files

@@files

end

end

# Make a new Computer instance:

hal = Computer.new("Dave", 12345)

puts "Current user: #{hal.current\_user}"

# @username belongs to the hal instance.

puts "Manufacturer: #{$manufacturer}"

# $manufacturer is global! We can get it directly.

puts "Files: #{Computer.display\_files}"

# @@files belongs to the Computer class.

**RUBY on Rails**

def create\_record(attributes, raise\_error = false)

record = build\_record(attributes)

yield(record) if block\_given?

saved = record.save

set\_new\_record(record)

raise RecordInvalid.new(record) if !saved && raise\_error

record

end

Ruby on Rails

Batman will win! Iron Man will

1. BThe rails new command created a new Rails app named MySite. It generated a number of files and folders that we will use to build the app. In the Code Editor, click on the folder icon to see these files and folders. We'll see what these files and folders are for in the next exercises. The rails new command is the starting point of every Rails project.
2. The bundle install command installed all the software packages needed by the new Rails app. These software packages are called gems and they are listed in the file **Gemfile**.
3. The rails server command started the Rails development server so that we could preview the app in the browser by visiting [http://localhost:8000](http://localhost:8000/). This development server is called WEBrick.

1. The rails generate model command created a new model named Message. In doing so, Rails created two files:

1. a model file in **app/models/message.rb**. The model represents a table in the database.
2. a migration file in **db/migrate/**. Migrations are a way to update the database.

2. Open the migration file in **db/migrate/**. The migration file contains a few things:

1. The change method tells Rails what change to make to the database. Here it uses the create\_table method to create a new table in the database for storing messages.
2. Inside create\_table, we added t.text :content. This will create a text column called content in the messages tables.
3. The final line t.timestamps is a Rails command that creates two more columns in the messages table called created\_at and updated\_at. These columns are automatically set when a message is created and updated.

3. The rake db:migrate command updates the database with the new messages data model.

4. Finally the rake db:seed command seeds the database with sample data from **db/seeds.rb**.

atman will win! Iron Man will win! win! Iron Man will win!