* **INTRODUCTION TO RUBY**
* **Overview & Sneak Peek**

Ruby is a powerful, flexible programming language you can use in web/Internet development, to process text, to create games, and as part of the popular Ruby on Rails web framework. Ruby is:

* **High-level**, meaning reading and writing Ruby is really easy—it looks a lot like regular English!
* **Interpreted**, meaning you don't need a compiler to write and run Ruby. You can write it here at Codecademy or even on your own computer (many are shipped with the Ruby interpreter built in—we'll get to the interpreter later in this lesson).
* **Object-oriented**, meaning it allows users to manipulate data structures called objects in order to build and execute programs. We'll learn more about objects later, but for now, all you need to know is ***everything in Ruby is an object.***
* **Easy to use**. Ruby was designed by Yukihiro Matsumoto (often just called "Matz") in 1995. Matz set out to design a language that emphasized human needs over those of the computer, which is why Ruby is so easy to pick up.
* **Data Types: Numbers, Strings, Booleans**

There are 3 **data types** in Ruby that we're interested in right now: **numbers**, **booleans** (which can be true or false), and **strings**(words or phrases like "I'm learning Ruby!"). It's also important to remember that Ruby is case-sensitive (it cares about capitalization).

* **Variables**

One of the most basic concepts in computer programming is the **variable**. You can think of a variable as a **word or name that grasps a single value.** For example, let's say you needed the number 25 from our last example, but you're not going to use it right away. You can set a variable, say my\_num, to grasp the value 25 and hang onto it for later use, like this:

my\_num = 25

***Declaring variables in Ruby*** is easy: you just write out a name like my\_num, use = to assign it a value, and you're done! If you need to **change a variable, no sweat: just type it again and hit = to assign it a new value.**

* **Math**

There are six arithmetic operators we're going to focus on:

**Addition (+)**

**Subtraction (-)**

**Multiplication (\*)**

**Division (/)**

**Exponentiation (\*\*)**

**Modulo (%)**

**Exponentiation raises one number (the base) to the power of the other (the exponent)**. For example, **2\*\*3**is 8, since 2\*\*3means "give me **2 \* 2 \* 2"** (2 multiplied together 3 times). **3\*\*2**is 9 (3 \* 3), and so on.

**Modulo** **returns the remainder of division**. For example, 25 % 7 would be 4, since 7 goes into 25 three times with 4 left over.

* **'puts' and 'print'**

The ***print command*** just takes whatever you give it and **prints it to the screen.**

***puts***(for "put string") is slightly different: it ***adds a new (blank) line after the thing you want it to print.*** You use them like this:

puts "What's up?"

print "Oxnard Montalvo"

***No parentheses or semicolons needed!***

* **Everything in Ruby is an Object**

Because **everything in Ruby is an object** (more on this later), **everything in Ruby has certain built-in abilities called methods**.

The **interpreter is the program that takes the code you write and runs it.** **You type code in the editor**, the **interpreter reads your code**, and it shows you the **result of running your code** in the **console.**

* **The '.length' Method**

**Methods are summoned using a “.”** . If you have a string, "I love espresso", and take the .length of it, Ruby will return the length of the string (that is, the number of characters—letters, numbers, spaces, and symbols).

"I love espresso".length

# ==> 15

Taking the length of input can be useful if, for example, you want to make a website that takes credit card payments. Ruby can check to make sure the credit card number appears to be valid.

puts 'Doncho Penev'.length

* **The '.reverse' Method**

**The*.reverse* method** is called the same way .length is; **spits out a backwards version of the string you gave it**. For instance,

"Eric".reverse

will result in

"cirE"

Reversing input can be useful if you want to **sort a list of values from highest to lowest instead of lowest to highest.**

puts 'Doncho Penev'.reverse

* **'.upcase' & '.downcase'**

the ***.upcase and .downcase* methods** **convert a string to ALL UPPER CASE or all lower case, respectively.**

puts "Doncho Penev".upcase

puts "Doncho Penev".downcase

* **Single-Line Comments**

***# sign*** is for **comments** in Ruby.

A comment is a bit of text that **Ruby won't try to run as code**:

The ***# sign*** should come before your comment and **affects anything you write after it, so long as you're on a single line**.

# I'm a full line comment!

"Eric".length # I'm a comment, too!

The **second example will return 4**, since the comment comes after the code that Ruby will execute.

* **Multi-Line Comments**

If you **start with *=begin***and **end with *=end*,** ***everything* between those two expressions will be a comment**.

=begin

I'm a comment!

I don't need any # symbols.

=end

Don't put **any space between the = sign and the words begin or end**. =begin and =end also need to be **on lines all by themselves, just as shown above.**

* **Naming Conventions**

By *convention*, **these variables should start with a *lowercase letter*** and ***words should be separated by underscores***, like counter and masterful\_method.

Ruby won't stop you from starting your local variables with other symbols, such as capital letters, $s, or @s, but by convention these mean different things, so it's best to avoid confusion

* **Variables & Data Types**

you ***declare a variable*** just by ***saying its name***, and you ***set it using =***

puts sum = 13+379

puts product = 923 \* 15

puts quotient = 13209 / 17

you call a method by using the .operator, like this***: "string".method***

***each method call on a separate line, or you can chain them together,*** like this:

name.method1.method2.method3

name = "Doncho"

puts name.downcase.reverse.upcase

* **PUTTING THE FORM IN FORMATTER**
* **What You'll Be Building**

This project will help you create a small program that will **read a user's input** and **correct his or her capitalization**. Users can provide an almost **infinite range of input,** so it makes our lives easier as programmers to **make their input standard before doing anything with it**.

script.rb

print "What's your first name? "

first\_name = gets.chomp

first\_name.capitalize!

print "What's your last name? "

last\_name = gets.chomp

last\_name.capitalize!

print "What city are you from? "

city = gets.chomp

city.capitalize!

print "What state or province are you from? "

state = gets.chomp

state.upcase!

puts "Your name is #{first\_name} #{last\_name} and you're from #{city}, #{state}!"

* Prompting the User

In order to get input from the user, we'll **first need to print a prompt on the screen**.

* **Getting Input**

variable\_name = gets.chomp

***gets***is the Ruby method that **gets input from the user**.

When getting input, **Ruby automatically adds a blank line** (or **newline**) **after each bit of input;**

***chomp* removes that extra line**. (Your program will work fine without chomp, but you'll get extra blank lines everywhere.)

print "What's your first name?"

first\_name = gets.chomp

print "What's your last name?"

last\_name = gets.chomp

print "Where do you live?"

city = gets.chomp

print "What state is that? Please use abbreviation."

state = gets.chomp

* **Printing the Output**

If **you define a variable monkey**that's equal to the **string "Curious George",** and then you have a string that says **"I took #{monkey} to the zoo",**

Ruby will do something called ***string interpolation***and **replace the #{monkey} bit** with the **value of monkey**

it will print "I took Curious George to the zoo".

first\_name = "Kevin"

puts "Your name is #{first\_name}!"

#will print "Your name is Kevin!"

puts "Your name is #{first\_name}#{last\_name}. You are from #{city},#{state}. "

#output: Your name is DoP. You are from Burgas,BS.

* **Formatting with String Methods**

print "This is my question?"

answer = gets.chomp

answer2 = answer.capitalize

answer.capitalize!

1. method, ***capitalize***, capitalises the **first letter of a string** and **makes the rest of the letters lower case.** We *assign the result to answer2*
2. The next line might look a little strange, we don't assign the result of capitalize to a variable. Instead you might notice t**he ! at the end of capitalize**. ***This modifies the value contained within the variable answer itself***. The next time you use the variable answer ***you will get the results of answer.capitalize***

print "What's your first name?"

first\_name = gets.chomp

first\_name.capitalize!

print "What's your last name?"

last\_name = gets.chomp

last\_name.capitalize!

print "Where do you live?"

city = gets.chomp

city.capitalize!

print "What state is that? Please use abbreviation."

state = gets.chomp

state.upcase!

puts "Your name is #{first\_name}#{last\_name}. You are from #{city},#{state}. "

* **CONTROL FLOW IN RUBY**
* **How It Works**

change their behavior in reaction to the **environment** (the collection of all variables and their values that exist in the program at a given time).

**Control flow** gives us the flexibility we're looking for. We can select different outcomes depending on information the user types, the result of a computation, or the value returned by another part of the program.

print "Integer please: "

user\_num = Integer(gets.chomp)

if user\_num < 0

puts "You picked a negative integer!"

elsif user\_num > 0

puts "You picked a positive integer!"

else

puts "You picked zero!"

end

* **If**

**Ruby's if statement** takes an **expression**, that **evaluates** to either **true or false**. If that expression is ***true,*** Ruby *executes the block of code that follows the if*. If it's not true (that is,***false***), Ruby *doesn't execute that block of code: it skips it and* goes on to the next thing.

if 1 < 2

print "I'm getting printed because one is less than two!"

end

Ruby doesn't care about **whitespace** (spaces and blank lines), so the indentation of the print statement isn't *necessary*. However, it's a convention that Rubyists (Ruby enthusiasts) follow, so it's good to get in the habit now. ***The block of code following an if should be indented two spaces.***

When you're ***done with your if***, you have to tell Ruby by **typing *end.***

if true

my\_name = 2

puts my\_name

end

* **Else**

if 1 > 2

print "I won't get printed because one is less than two."

else

print "That means I'll get printed!"

end

* **Elsif**

if x < y

# Assumes x and y are defined

puts "x is less than y!"

elsif x > y

puts "x is greater than y!"

else

puts "x equals y!"

end

* **Unless**

Sometimes you want to use **control flow to check if something is *false***, rather than if it's true.

Let's say you don't want to eat ***unless*** you're hungry. That is, while you're not hungry, you write programs, but if you *are* hungry, you eat. You might write that program in Ruby like this:

unless hungry

# Write some sweet programs

else

# Have some noms

end

hungry = false

unless hungry

puts "I'm writing Ruby programs!"

else

puts "Time to eat!"

end

* **Equal or Not?**

In Ruby, we **assign values to variables using =,** the **assignment operator**.

***we use ==, which is a comparator(also called a relational operator). == means "is equal to."*** When you type

x = 2

y = 2

if x == y

print "x and y are equal!"

end

you're saying: "if x equals y, print 'x and y are equal!'" You can also check to see if two values are **not equal using the != comparator.**

is\_true = 2 != 3

is\_false = 2 == 3

* **Less Than or Greater Than**

Those operators look like this:

* Less than: <
* Less than or equal to: <=
* Greater than: >
* Greater than or equal to: >=

test\_1 = 17 > 16

test\_2 = 21 < 30

test\_3 = 9 <= 9

test\_4 = -11 < 4

* **And**

You can also use **logical** or **boolean operators**.

Ruby has three**: and (&&), or (||), and not (!);** result in boolean values: **true or false.**

The boolean operator **and**, &&, only results **in true** when **both** expression on **either side of && are true.**

true && true

# => true

true && false

# => false

false && true

# => false

false && false

# => false

boolean\_2 = true && 100 >= 100

* **Or**

the **or** operator (||), called an **inclusive or** because it evaluates to **true when one or the other *or both* expressions are true**.

true || true

# => true

true || false

# => true

false || true

# => true

false || false

# => false

boolean\_1 = 2\*\*3 != 3\*\*2 || true

* **Not**

the boolean operator **not** (!). ! makes true values false, and vice-versa.

!true

# => false

!false

# => true

* **Combining Boolean Operators**

combine boolean operators in your expressions.

Combinations like (x && (y || w)) && z

Expressions in parentheses are always evaluated before anything outside parentheses.

boolean\_3 = true || !(true || false)

* **If, Else, and Elsif**

a = 10

b = 11

if a < b

print "a is less than b!"

elsif b < a

print "b is less than a!"

else

print "b is equal to a!"

end

if expression

# Do something

elsif expression

# Do something else

else

# Do yet another thing

end

* **Unless**

It will **do whatever you ask**unless the ***condition is true.*** In our example, problem is false, so we don't have a problem. We print Good to go!

unless condition

# Do something!

end

* **Billions of Booleans**

( 1 == 1 ) && ( 2 == 2 )

# true

( 1 == 2 ) || ( 2 == 2 )

# true

!( false )

# true

1. With && **both comparisons on the left and right must evaluate to true**for the entire statement to return true.
2. With || **either the right or left side must evaluate to true.** If the left side evaluates to true, ***the right side will not be tried.***
3. With ! **you reverse the result**.

* **What You'll Be Building**

In this project, we'll combine control flow with a few new Ruby string methods to Daffy Duckify a user's string, replacing each "s" with "th".

print "Thtring, pleathe!: "

user\_input = gets.chomp

user\_input.downcase!

if user\_input.include? "s"

user\_input.gsub!(/s/, "th")

else

puts "Nothing to do here!"

end

puts "Your string is: #{user\_input}"

* **Getting User Input**

First, we should print a statement to prompt the user for input, then set that input to a variable using gets.chomp.

* **Downcase!**

We want to make sure we capture both "S" and "s" in the user's input. We could write separate if/ else statements to handle this, but we can also **use .downcase! to convert the user's input to all lower case.** That way, we only have to search for "s".

* **Setting Up the 'If' Branch, Part 1**

For the if half of our branch, we want to **check whether the user's input contains an "s".**

***if string\_to\_check.include? "substring"***

We can do that using **Ruby's .include? method**, which evaluates to **true if it finds what it's looking for** and false otherwise.

(As a general rule, **Ruby methods that end with ?evaluate to the boolean values true or false**.)

print "Tell me something: "

user\_input = gets.chomp

user\_input.downcase!

if user\_input.include? "s"

print "There is an \"s\""

end

* **Setting Up the 'If' Branch, Part 2**

When we find "s", we want Ruby to ***replace every instance of "s" it finds with "th". We can do this with the .gsub! method,*** which stands for ***g****lobal****sub****stitution.*

string\_to\_change**.gsub!(/s/, "th")**

When we get to later lessons, we'll explain why the /s/ has to be between slashes instead of between quotes. Note: you *cannot* put a space between gsub! and the bit in parentheses.

Remember, you want **the ! at the end of the method name** so that **Ruby will change the string in-place.**

* **Setting Up the 'Else' Branch**

Now we just need to let the user know if **we don't find any instances of the letter** "s".

print "Tell me something: "

user\_input = gets.chomp

user\_input.downcase!

if user\_input.include? "s"

user\_input.gsub!(/s/, "th")

else

puts "No \"s\"'s in the your string"

end

* **Returning the Final String—Er, "Thtring"**

Home stretch—now we want to display the Daffy Duckified string to the user. You can do that using the **string interpolation** we learned earlier:

my\_string = "muchachos"

print "Adios, #{my\_string}!"

# ==> "Adios, muchachos!"

print "Tell me something: "

user\_input = gets.chomp

user\_input.downcase!

if user\_input.include? "s"

user\_input.gsub!(/s/, "th")

puts "Did you say: #{user\_input}"

else

puts "No \"s\"'s in the your string"

end

* **LOOPS & ITERATORS**
* **The 'While' Loop**

**repeat an action in Ruby while a certain condition is true,** but you **don't know how many times**.

A good example would be prompting a user for a certain type of input: if they insist on giving you the wrong thing, you may have to re-ask them several times before you get the kind of input you're looking for.

**a while loop checks to see if a certain condition is true,** and while it is, the **loop keeps running**. As soon as the **condition stops being true, the loop stops!**

counter = 1

while counter < 11

puts counter

counter = counter + 1

end

* **Danger: Infinite Loops!**

What if we'd forgotten to increment counter? It would have stayed at 1, the loop would have kept checking to see if it was less than 11 (and 1 is always less than 11), **and the loop would never have ended.** This is called an ***infinite loop***and it will cause your programs to crash

i = 0

while i < 5

puts i

i=i+1

end

* **The 'Until' Loop**

The complement to the while loop is the untilloop. It's sort of like a backward while:

i = 0 until i == 6 i = i + 1 end puts i

1. In the example above, we first create a variable i and set it to 0 (zero).
2. Then we execute a block of code until i is equal to 6. That block of code increments i.
3. When i is equal to 6, the block ends.
4. Finally, we print 6, the value of i, to the console.

counter = 1

until counter > 10

puts counter

# Add code to update 'counter' here!

counter = counter + 1

end

* **More Assignment Operators**

 counter = counter + 1,

A shortcut is to **use an *assignment operator*.**

**assignment operator: =** **sets a variable.**

**update a variable** with additional assignment operators

**include +=, -=, \*=, and /=**.

counter += 1

counter = 1

while counter < 11

puts counter

counter += 1

end

* **The 'For' Loop**

Sometimes **you do know how many times you'll be looping**,

for num in 1...10

puts num

end

* **Inclusive and Exclusive Ranges**

for num in 1...10.

What this says to Ruby is: "**For the variable num**in the **range 1 to 10**, *do the following."* The following was to puts "#{num}", so as num took on the values of 1 to 9, one at a time, those values were printed to the console.

we used **three dots (3)** in the range; this tells Ruby to **exclude the final number** in the count: for num in 1…10 means **“go up to but don’t include 10.”** If we use **two dots (2)**, this tells Ruby to **include the highest number in the range**.

for num in 1..15

puts num

end

for num in 1..20

puts num

end

* **The Loop Method**

In this case, it's also possible to **repeat an action using an *iterator***.

An **iterator** is just a **Ruby method** that **repeatedly invokes a block of code**.

The simplest iterator is the **loop method**. You can create a **basic (but infinite!) loop** by simply typing

loop { print "Hello, world!" }

In Ruby, **curly braces ({}) are generally interchangeable** with the keywords **do (to open the block) and end (to close it).** Knowing this, we can write a smarter loop than the one above:

i = 0

loop do

i += 1

print "#{i}"

break if i > 5

end

**The break keyword**: **breaks a loop as soon as its condition is met.**

i = 20

loop do

i -= 1

print "#{i} "

break if i <= 0

end

* **Next!**

The **next keyword** can be used to **skip over certain steps in the loop**. For instance, if we don't want to print out the even numbers, we can write:

for i in 1..5

next if i % 2 == 0

print i

end

1. In the above example, we loop through the range of 1 through 5, assigning each number to i in turn.
2. If the remainder of  i / 2 is zero, **we go to the next iteration of the loop**.

i = 20

loop do

i -= 1

next if i % 2 != 0

print "#{i}"

break if i <= 0

end

* **Saving Multiple Values**

to **save a range of numbers in a variable**.

In Ruby, we can **pack multiple values into a single variable using an *array***. An array is just **a list of items between square brackets**, like so: **[1, 2, 3, 4].**

my\_array = [1,2,3,4,5]

* **The .each Iterator**

A more useful **iterator is the .each method**, which **can apply an expression to each element of an object, one at a time**. The syntax looks like this:

object.each { |item|

# Do something

}

You can also use the do keyword instead of {}:

object.each do |item|

# Do something

end

The **variable name between | |**can be anything you like: it's just a **placeholder for each element of the object you're using .each**on.

array = [1,2,3,4,5]

array.each do |x|

x += 10

print "#{x} "

end

* **Try It Out!**

numbers = [1, 2, 3, 4, 5]

# one way to loop

numbers.each { |item|

puts item

}

# another way to loop

numbers.each do |item|

puts item

end

odds = [1,3,5,7,9]

odds.each {|x|

x \*=2

print "#{x} "

}

odds.each do |odd|

print odd\*2

end

* **The .times Iterator**

The **.times method**: can **perform a task** on **each item in an object** ***a specified number of times.***

For example, if we wanted to print out "Chunky bacon!" ten times, we might type

10.times { print "Chunky bacon!" }

7.times {print "Hi. "}

* **Looping with 'While'**

i = 3

while i > 0 do

print i

i -= 1

end

1. In the above example, we **create a variable called i and set it to 3.**
2. Then, we print out 321 since we execute the loop as long as i is positive.

i = 1

while i <= 50 do

print i

i += 1

end

* **Looping with 'Until'**

i = 3

while i > 0 do

print i

i -= 1

end

j = 3

until j == 0 do

print j

j -= 1

end

In the example above, we wrote the same loop using while and using until.

i = 1

until i > 50 do

print i

i += 1

end

* **Looping with 'For'**

for k in 1..3

print k

end

In the above example, we print out 123 by virtue of looping from 1 to 3 inclusive.

i = 1

for i in 1..50

print i

i += 1

end

for variable in (range)

# Do something end

* **Loop the Loop with Loop**

m = 0

loop do

m += 1

print m

break if m == 10

end

In the example above, we print out 12345678910 since we loop 10 times.

loopindex=1

loop {

print "Ruby!"

loopindex += 1

break if loopindex > 30

}

* **Iterating with .times**

30.times { print "Ruby! "}

* **REDACTED!**
* **What You'll Be Building**

**Hiding information** is a major part of programming: protecting passwords, establishing secure connections, and securing programs against tampering all rely on **controlling access to information.**

While we won't be able to really dig into information hiding until after we cover *hashes* in a later course, we can write a simple program to **change a user's input with the tools we have now: arrays and iterators.**

puts "Text to search through: "

text = gets.chomp

puts "Word to redact: "

redact = gets.chomp

words = text.split(" ")

words.each do |word|

if word != redact

print word + " "

else

print "REDACTED "

end

end

* **Getting the User's Input**

First things first: we'll need to **get the user's input.**

**Instructions**

Use puts to prompt the user for input two times. For the first puts, declare a variable called **text**and set it equal to the user's input via **gets.chomp.**

For the second puts, declare a variable called **redact**and set it equal to the **user's input using gets.chomp.**

* **The .split Method**

Next, we'll want to **divide the user's input into individual words.**

Ruby has a built-**in method for this called .split**; it **takes in a string** and **returns an array**.

If we ***pass it a bit of text in parentheses***, **.split will divide the string** wherever it ***sees that bit of text, called a delimiter***.

For example,

text.split(",")

tells Ruby to **split up the string text whenever it sees a comma**.

words = text.split(" ")

we get an array made up of the words from text.

* **Redacted!**

**our iterators to go through the user's text.**

letters = ['a', 'b', 'c', 'd']

letters.each do |letter|

print letter

end

The example above just serves as a reminder of using .each on the letters array.

array.each { |placeholder| #action }

array.each do |placeholder|

# action

# another action

# yet another action!

end

* **Control Flow Know-How**

Good! There were two problems with our output, though: we didn't have spaces between our words, and our program didn't actually replace the word we wanted to redact with the word "REDACTED".

if var == 10

print "Variable is 10"

else

print "Variable is something else"

end

We can fix that with some if/else magic! The above example just reminds you how an if/else block works.

puts "Give me the text, please?"

text = gets.chomp

puts "What do you desire redacted?"

redact = gets.chomp

words = text.split(" ")

words.each{ |word|

if word == redact

print "REDACTED "

else

print word + " "

end

}

* **DATA STRUCTURES**
* **Creating Arrays**

an **array can be used to store a list of values in a single variable.**

demo\_array = [100, 200, 300, 400, 500]

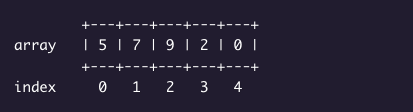
* **Access by Index**

**each element in the array has what's called an *index*.** The **first element** is at **index 0**, the next is at index 1, the following is at index 2, and so on. We can **access elements of the array directly through these numbers using brackets**, like so:

array = [5, 7, 9, 2, 0]

array[2]

# returns "9", since "9" # is at index 2



We can access the ith element of an array called array by putting the index in square brackets, like so***:****access by index*.

***array[i].***

demo\_array = [100, 200, 300, 400, 500]

print demo\_array[2]

* **Arrays of Non-Numbers**

**you can make an array of *any collection of Ruby objects*** :array of **Booleans**; array of **strings**

string\_array = ["hello","my","friend"]

* **Arrays of Arrays**

**array of arrays** *= multidimensional* arrays

the act of **adding more arrays expands the array out of its string-like shape**. For instance, the array in the editor is a **two-dimensional array.**

multi\_d\_array = [[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]

multi\_d\_array.each { |x|

puts "#{x}\n"

}

* **Create Your Own**

See how a two-dimensional array with the same number of elements per row and overall rows is a square? An array (like a line) is one-dimensional; an array of arrays (like a square) is two-dimensional.

my\_2d\_array = [[1,false,3],[22,true],[3,"hi",2,6]]

* **Introduction to Hashes**

We know that **arrays are indexed with numbers that start with 0** and go up to the **array's length minus one**.

But what if we **want to use numeric indices that don't go in order from 0 to the end of the array**? What if we **don't want to use numbers as indices** at all?

We'll need a **new array structure called a *hash*.**

Hashes are sort of like **JavaScript objects or Python dictionaries**. A **hash is a collection of *key-value pairs*.** Hash syntax looks like this:

hash = {

key1 => value1,

key2 => value2,

key3 => value3

}

**Values are assigned to keys using =>.**

You can use **any Ruby object for a key or value**.

my\_hash = {

"name" => "Eric",

"age" => 26,

"hungry?" => true

}

puts my\_hash["name"]

puts my\_hash["age"]

puts my\_hash["hungry?"]

* **Using Hash.new**

What we just showed you was **hash *literal notation***. ; you **literally describe what you want in the hash**: you **give it a name** and you **set it equal to one or more key => value pairs** inside **curly braces.**

You can also create a hash using Hash.new, like so:

my\_hash = Hash.new

Setting a **variable equal to Hash.new creates a new, empty hash**; it's the same as setting the variable equal to **empty curly braces ({}).**

pets = Hash.new

* **Adding to a Hash**

We can **add to a hash two ways**:

1. if we **created it using literal notation**, we can simply **add a new key-value pair directly between the curly braces**.
2. If we **used Hash.new**, we can **add to the hash using bracket notation**:

pets = Hash.new

pets["Stevie"] = "cat"

# Adds ***the key "Stevie****"* with the ***value "cat"* to the hash**

pets = Hash.new

pets["boolea"] = true

pets["another"] = "One"

pets["numbersss"] = 7

* **Accessing Hash Values**

You can **access values in a hash just like an array**.

pets = {

"Stevie" => "cat",

"Bowser" => "hamster",

"Kevin Sorbo" => "fish"

}

puts pets["Stevie"]

# will print "cat"

1. In the example above, we **create a hash called pets.**
2. Then we **print cat**by ***accessing the key "Stevie" in the pets hash.***

pets = Hash.new

pets["Luna"] = "cat"

puts pets["Luna"]

* **(Re)Introduction to Iteration**

When we **loop over an array or a hash**, we say that we ***iterate* over it.**

We'll be using **the .each iterator** to **iterate over arrays** and **hashes** in this section.

friends = ["Milhouse", "Ralph", "Nelson", "Otto"]

family = {

"Homer" => "dad",

"Marge" => "mom",

"Lisa" => "sister",

"Maggie" => "sister",

"Abe" => "grandpa",

"Santa's Little Helper" => "dog"

}

friends.each { |x|

puts "#{x}"

}

family.each { |x, y|

puts "#{x}: #{y}"

}

* **Iterating Over Arrays**

numbers = [1, 2, 3, 4, 5]

numbers.each { |element|

puts element

}

1. In the example above, we create an array called numbers with 5 elements.
2. Then we say, "Take this array and for each element, print it to the console." As usual, we can use any placeholder name for the bit between two | | characters.

languages = ["HTML", "CSS", "JavaScript", "Python", "Ruby"]

languages.each { |element|

puts element

}

* **Iterating Over Multidimensional Arrays**

**iterate over a multidimensional array.**

**2-D array, s**; want to **iterate over s**in such a way that **we don't print out each element as an array,** like ["ham", "swiss"], but **each element within each sub-array**, so we get a list of all the meats and cheeses within s.

If we just wanted to access "swiss", we could type

**s[0][1]**

Meaning, "bring me the second element of the first element," which is "swiss." If we iterate over a regular array using

array.each { |element|

action

}

s = [["ham", "swiss"], ["turkey", "cheddar"], ["roast beef", "gruyere"]]

s.each { |sub\_array|

sub\_array.each { |element|

puts element

}

}

* **Iterating Over Hashes**

When **iterating over hashes**, we **need *two placeholder variables* to represent each *key/value pair.***

restaurant\_menu = {

"noodles" => 4,

"soup" => 3,

"salad" => 2

}

restaurant\_menu.each do |item, price|

puts "#{item}: #{price}"

end

1. In the example above, we **create a new hash called *restaurant\_menu***.
2. Then, we **loop through the restaurant\_menu hash** and **assign the key to item** and **the value to price for each iteration.**
3. Finally, we puts out:

noodles: 4

soup: 3

salad: 2

secret\_identities = {

"The Batman" => "Bruce Wayne",

"Superman" => "Clark Kent",

"Wonder Woman" => "Diana Prince",

"Freakazoid" => "Dexter Douglas"

}

secret\_identities. each{ |key,value|

puts "#{key}: #{value}"

}

my\_array = [["new","old",16],[23, 24, true],["hi","bye"]]

* Hashes

Good! Now let's create a hash. Feel free to use either hash literal notation or Hash.new.

prices = { "apple" => 0.52, "banana" => 0.23, "kiwi" => 1.42 } sounds = Hash.new sounds["dog"] = "woof" sounds["cat"] = "meow"

my\_hash = {

"bed" => true,

"type" => "modern",

"vacant\_date" => 27

}

my\_hash = Hash.new

my\_hash["bed"] = true

my\_hash["type"] = "modern"

my\_hash["vacant\_date"] = 27

puts my\_hash["bed"]

hash\_name = Hash.new # use bracket notation to assign a new key-value pair hash\_name[key1] = value1

Literal notation:

hash\_name = {

key1 => value1,

key2 => value2,

key3 => value3

}

* **Iterating Over a Hash**

numbers = [1, 2, 3, 4, 5]

numbers.each { |element|

puts element

}

lunch\_order = {

"Ryan" => "wonton soup",

"Eric" => "hamburger",

"Jimmy" => "sandwich",

"Sasha" => "salad",

"Cole" => "taco"

}

lunch\_order.each { |x,y|

puts "#{y}"

}

* **CREATE A HISTOGRAM**
* **What You'll Be Building**

In this project, we'll write a program that **takes a user's input**, then **builds a hash from that input. Each key** in the hash will be **a word from the user;** **each value** will be the **number of times that word occurs**.

For example, if our program gets the string "the rain in Spain falls mainly on the plain," it will return

the 2

falls 1

on 1

mainly 1

in 1

rain 1

plain 1

Spain 1

A **visual representation of data** like this is **called a *histogram*.**

puts "Text please: "

text = gets.chomp

words = text.split(" ")

frequencies = Hash.new(0)

words.each { |word|

frequencies[word] += 1

}

frequencies = frequencies.sort\_by {|a, b|

b

}

frequencies.reverse!

frequencies.each { |word, frequency|

puts word + " " + frequency.to\_s

}

* **Building the Words Array**

Next, we'll want to **turn the user's string** into **something we can iterate over** - an array!

**calling *the .split*method on text, we can transform it into an array.**

* **Creating the Frequencies Hash**

start **counting words using a hash**; make sure the **hash has a *default value*.**

h = Hash.new("nothing here")

puts h

# {}

puts h["kitty"]

# nothing here

1. In the example above, we **create a new, empty hash h** that has a ***default value* of "nothing here"**.
2. Then we **print out {},** the **value of h**, **just to show that h really is empty**.
3. Then we **print out nothing here**as we t**ry to access the value stored by the key "kitty".**

If you have a **hash with a *default value***, and you **try to access a *non-existent key****,* you ***get that default value.***

* **Iterating Over the Array**

we want to **iterate over words** to **add each word** to our **frequencies hash**, **one at a time.**

colors = {

"red" => 2,

"blue" => 3

}

colors["blue"] += 1

puts colors["blue"]

1. In the above example, **we first create a hash mapping strings to integers**.
2. Then, we **increment the value stored by "blue"by 1.**
3. Finally, we print out 4, the **value stored by "blue".**

Use **.each to iterate over the words array**.

For **each word we find**, assume that the **word itself is a key in frequencies**and **increment its value by 1.**

This is why ***our default is 0***. The **first time we find the word**, it **will have a default value of 0** that we can increment by1`.

puts "Enter a phrase you'd like to analyze: "

text = gets.chomp

words = text.split

frequencies = Hash.new(0)

words.each { |word|

frequencies[word] += 1

}

* **Sorting the Hash**

We have a **hash full of word / frequency key-value pairs**.

Now we need to figure out a way to get our information in the order we want it.

colors = {

"blue" => 3,

"green" => 1,

"red" => 2

}

colors = colors.sort\_by do |color, count|

count

end

colors.reverse!

1. In the example above, we **first create a hash** called **colors that maps color strings to numbers.**
2. Then, we **sort colors**into green, red, and blue, **from smallest to largest by *count****.* Just so you know, the **.sort\_by function returns an array of arrays**
3. Finally, we **reverse the array order** so that the **colors with the largest counts are first**.

Use **.sort\_by to sort the frequencies hash by word count** and **store the result back in frequencies.**

Use .**reverse! to reverse the sorted frequencies array**.

Our first step should be to **reassign frequencies to its sorted version** **(.sort\_by**doesn't sort the hash in-place—it **will create a *copy* that is sorted**):

frequencies = frequencies.sort\_by { |k, v|

v

}

This **actually returns an *array* of values**, which you can **then reverse in-place with reverse!:**

frequencies.reverse!

* **Iterating Over the Hash**

, we'll need to i**terate over the array to print out each key-value pair to the console.**

fruit = {

"apple" => 2,

"banana" => 3,

"cherry" => 5

}

fruit.each do |name, count|

puts name + " " + count.to\_s

end

1. In the example above, we create a hash called fruit that maps names of fruit to the amount that we own.
2. Then, we iterate **over .each key/value pair**, storing the key as name and the value as count.
3. Finally, **we print out the key and value separated by a space**. Note that we must first **convert the value from a number to a string** using **.to\_s**

be**fore we can concatenate it.**

Method .to\_s

converts the value from a number to a string

puts "Enter a phrase you'd like to analyze: "

text = gets.chomp

words = text.split

frequencies = Hash.new(0)

words.each { |word|

frequencies[word] += 1

}

frequencies = frequencies.sort\_by{ |word, count|

count

}

frequencies.reverse!

frequencies.each{ |key, value|

puts "#{key} #{value}"

}

* **METHODS, BLOCKS, & SORTING**
* **Why Methods?**

A **method** is a **reusable section of code** written to **perform a specific task in a program.**

A few good reasons to divide your programs into methods:

1. it's much **easier to find and fix bugs** if you've **organized your program well**.
2. By **assigning** **specific tasks to separate methods** (an idea computer scientists call **separation of concerns**), program *less redundant* and your *code more reusable*—not only can you repeatedly **use the same method in a single program** without rewriting it each time, but you can even use that **method in *another* program.**
3. When we learn more about **objects**, you'll find out there are a lot of interesting things we can do with methods in Ruby.

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)

* **Method Syntax**

**Methods are defined** using the **keyword *def***(short for "define").

Methods have three (3) parts:

1. **The *header***, which includes **the def keyword**, **the name of the method**, and **any *arguments* the method takes**.
2. **The *body***, which is **the code block** that describes the p**rocedures the method carries out**; indented **two spaces by convention**
3. The method **ends with the end keyword.**

a simple function, welcome, that just prints "Welcome to Ruby!" to the console:

def welcome

puts "Welcome to Ruby!"

end

def puts\_1\_to\_10

(1..10).each { |i|

puts i

}

end

puts\_1\_to\_10 # Ignore this for now. We'll explain it soon!

# Define your method below!

def greeting

puts "Greetings!"

end

# Define your method above this line.

greeting # Ignore this for now. We'll explain

# it in the next exercise!

* **Call It!**

*call* it, or tell your program to execute it.

If the **program doesn't find a method** called cartoon\_fox, it will **return a *NameError****.*

You **call a method** just **by typing its name**.

def array\_of\_10

puts (1..10).to\_a

end

array\_of\_10

Method .to\_a means to Array, enters the values as elements of an array

* **Parameters and Arguments**

If a **method takes arguments**, we say it ***accepts* or *expects* those arguments**. We might **define a function**, square, like so:

def square(n)

puts n \*\* 2

end

and call it like this:

square(12)

# ==> prints "144"

The ***argument* i**s the piece of code you actually **put between the method's parentheses** **when you *call it,*** and the ***parameter***is the **name you put between the method's parentheses when you *define***it.

**we gave it the parameter n**(for "number") and **passed it the argument 12**when we called it.

**parameters are placeholders** the **method definition gives to arguments**

***Parentheses are usually optional in Ruby***, ***parameters and arguments in parentheses for readability.***

def cubertino(n)

puts n \*\* 3

end

cubertino(8)

* **Splat!**

Let's say you have a **method,** friend, that **puts the argument it receives from the user**:

def friend(name):

puts "My friend is " + name + "."

end

***splat arguments*. Splat arguments are arguments *preceded by a \*,*** which tells the program that the **method can receive one or more arguments**.

def what\_up(greeting, \*friends)

friends.each { |friend| puts "#{greeting}, #{friend}!" }

end

what\_up("What up", "Ian", "Zoe", "Zenas", "Eleanor")

* **Let's Learn Return**

Sometimes we **don't just want a method to print something to the console**, but we actually want that **method to hand us (or another method!) back a value**.

def double(n)

return n \* 2

end

output = double(6)

output += 2

puts output

1. In the example above, we define a new method called double that accepts one argument called n.
2. Inside the method, we return two times n.
3. After that, we **call our new double method with an argument of 6 and store the result, 12, in output.**
4. Then, we increase output to 14 and print it out to the console.

def add (number1,number2)

return number1+number2

end

* **Practice Makes Perfect**

def by\_five?(n)

return n % 5 == 0

end

**on how to define a method.**

def greeter (name)

return "Hi, " + name

end

def by\_three? (number)

if number%3==0

return true

else

return false

end

end

* **Blocks Are Like Nameless Methods**

Most methods that you've worked with have defined names that either you or someone else gave them (*i.e.* [array].sort(), "string".downcase(), and so on). You **can think of blocks as a way of creating methods that don't have a name**. (These are similar to **anonymous functions in JavaScript** or lambdas in Python.)

**Blocks can be defined with either the keywords do and end**or with **curly braces ({}).**

1.times do

puts "I'm a code block!"

end

1.times { puts "As am I!" }

* **How Blocks Differ from Methods**

**differences between blocks and methods**,

however.

The capitalize method capitalizes a word, and we can continually invoke the capitalize method by name. We can capitalize("matz"), capitalize("eduardo"), or any string we like to our hearts' content.

However, the block that we define (following .each) will only be called *once*, and in the context of the array that we are iterating over. It appears just long enough to do some work for each, then vanishes into the night.

# method that capitalizes a word

def capitalize(string)

puts "#{string[0].upcase}#{string[1..-1]}"

end

capitalize("ryan") # prints "Ryan"

capitalize("jane") # prints "Jane"

# block that capitalizes each string in the array

["ryan", "jane"].each {|string| puts "#{string[0].upcase}#{string[1..-1]}"} # prints "Ryan", then "Jane"

* **Using Code Blocks**

A **method can take a block as a parameter.**

That's what .each has been doing this whole time: taking a block as a parameter and doing stuff with it!

**Passing a block to a method** is a great way of ***abstracting***certain tasks from the **method and defining those tasks when we call the method**. Abstraction is an important idea in computer science, and you can think of it as meaning **"making something simpler."**

Just like saying "house**" simplifies listing its components**, **using a block to defin**e the task **you want the method (like .each)** to do simplifies the task at hand.

# The block, {|i| puts i}, is passed the current

# array item each time it is evaluated. This block

# prints the item.

[1, 2, 3, 4, 5].each { |i| puts i }

# This block prints the number 5 for each item.

# (It chooses to ignore the passed item, which is allowed.)

[1, 2, 3, 4, 5].each { |i| puts i\*5 }

* **Introduction to Sorting**

**Sorting arrays** is a very common problem in computer science. There are **many *algorithms***

how to use **Ruby's built-in sorting library** (which implements a few of these algorithms).

***.sort***would sort a **copy of my\_array**, while **.sort! sorts my\_array in-place** (that is, it ***destroys the original array*** and ***replaces it with the sorted version***).

my\_array = [3, 4, 8, 7, 1, 6, 5, 9, 2]

# Call the sort! method on my\_array below.

# my\_array should then equal [1, 2, 3, 4, 5, 6, 7, 8, 9].

my\_array.sort!

puts my\_array

* **Foundations**

**Most sorting algorithms assume we are sorting an array of items**, which **involves comparing any two items** in the array and **deciding which** should come first.

# library sorting code

books = ["Charlie and the Chocolate Factory", "War and Peace", "Utopia", "A Brief History of Time", "A Wrinkle in Time"]

# How might we sort! the books in alphabetical order?

books.sort!

puts books

* **The Combined Comparison Operator**

We can also use a **new operator** called the ***combined comparison operator***to **compare two Ruby objects**. The **combined comparison operator** looks like this:**<=>.**

It returns **0 if the first *operand***(item to be compared) **equals the second**, **1 if the first operand is greater than the second**, and**-1 if the first operand is less than the second.**

***A block that is passed into the sort method*** must **return either1, 0, or -1**. It should return -1 if the **first block parameter should come before the second**, 1 if vice versa, and 0 if they are **of equal weight**, meaning one does not come before the other (*i.e.* if two values are equal)

book\_1 = "A Wrinkle in Time"

book\_2 = "A Brief History of Time"

book\_1 <=> book\_2

* **Getting Technical**

What if we wanted to **sort the books by title, but from Z – A,** or descending order?

The **sort method assumes by default** that you want to sort in **ascending order**, but it **accepts a block as an optional argument** that allows you, the programmer, to **specify how two items should be compared.**

books.sort do |first, second|

if first < second # first book before second alphabetically

-1

elsif first > second # first after second

1

else # first and second are the same

0

end

end

books = ["Charlie and the Chocolate Factory", "War and Peace", "Utopia", "A Brief History of Time", "A Wrinkle in Time"]

# To sort our books in ascending order, in-place

books.sort! { |firstBook, secondBook| firstBook <=> secondBook }

# Sort your books in descending order, in-place below

books.sort! { |firstBook, secondBook| secondBook <=> firstBook }

**Basic Methods**

Let's quickly review how to create a basic Ruby method.

def double(n) return n \* 2 end

The example above is just a syntax reminder.

def double(n)

return n \* 2

end

def method\_name(optional arguments)

# Do something

end

def welcome

puts "Welcome to Ruby!"

end

welcome

**Methods With Arguments**

Good! Now let's make our method a bit more complex by adding arguments and a returnstatement.

def double(n)

return n \* 2

end

def welcome (name)

return "Hello, #{name}"

end

puts welcome("Don")

**Blocks**

numbers = [5, 2, 8]

sum = 0

numbers.each do |n|

sum += n

end

puts sum

object.method { |placeholder| action }

my\_array = [1, 2, 3, 4, 5]

my\_array.each do |x|

puts x\*\*2

end

**Sorting**

Finally, let's review what we learned about sorting.

books.sort! do |firstBook, secondBook| firstBook <=> secondBook end

Remember that the above example was how we sorted in alphabetical order.

to sort alphabetically, we did this:

books.sort! { |firstBook, secondBook| firstBook <=> secondBook }

Sorting in descending alphabetical order:

fruits = ["orange", "apple", "banana", "pear", "grapes"]

fruits.sort! do |firstBook, secondBook|

secondBook <=> firstBook

end

**ORDERING YOUR LIBRARY**

**What You'll Be Building**

We noticed in the last lesson that .sort! didn't have a built-in way of handling sorting in reverse alphabetical order. Now that we know how to write our own Ruby methods, we can fix that!

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)}"

**Default Parameters**

Let's start with the new bit of code you saw in exercise 1:

def alphabetize(arr, rev=false)

The first part makes sense—we're defining a method, alphabetize. We can guess that the first parameter is an array, but what's this rev=false business?

What this does is tell Ruby that alphabetize has a second parameter, rev (for "reverse") that will ***default* to false**if the user doesn't type in two arguments. You might have noticed that our first call to alphabetize in exercise 1 was just

alphabetize(books)

Ruby **No second argument passed**, but **second parameter defined** with method def : **didn't see a rev, so it gave it the default value of false.**

**Sorting**

Great! Now let's add a little logic to our method.

numbers = [5, 1, 3, 8] numbers.sort! puts numbers

1. In the above example, we create a new array called numbers.
2. Then, we sort the array.
3. Finally, we print out 1, 3, 5, 8, the sorted array.

In Ruby, there are two sorting methods, .sort or sort!. The first method, .sort, simply returns a sorted array while leaving the original array alone. The second method, .sort!, modifies the actual array.

Inside your method, add a line that calls .sort! on the arr array. Since it is the last line of the method, the sorted array will be returned.

def alphabetize (arr, rev=false)

arr.sort!

end

numbers = [1,3,5,78,32,45]

puts alphabetize(numbers)

**Sorting With Control Flow**

Great! Now we need to add the right logic to our method.

numbers = [1, 2, 3, 4, 5] numbers.reverse! puts numbers

1. In the example above, we create an array called numbers.
2. Then, we reverse the array. Like with .sort!, the exclamation mark means we modify the actual array.
3. Finally, we print out 5, 4, 3, 2, and 1.

def alphabetize (arr, rev=false)

arr.sort!

if rev == true

arr.reverse!

else

return arr

end

end

numbers = [1,3,5,78,32,45]

puts alphabetize(numbers)

**HASHES AND SYMBOLS**

* **The Story So Far**

Recall that **hashes are**[**collections**](https://www.codecademy.com/courses/ruby-beginner-en-F3loB/1#!/exercises/0)**of key-value pairs**, where **a unique key** is associated with **some value**. For example:

breakfast = {

"bacon" => "tasty",

"eggs" => "tasty",

"oatmeal" => "healthy",

"OJ" => "juicy"

}

Remember that **keys must be unique**, but values can repeat.

We can **create hashes several ways**, but two of the most popular are

1. **hash literal notation**:

new\_hash = { "one" => 1 }

1. **hash constructor notation**:

new\_hash = Hash.new

**Iterating Over Hashes**

We can also iterate over hashes using the .eachmethod. For example, we could do

my\_hash.each do |key, value| puts my\_hash[] end

This will print out a list of keys and values from my\_hash, each on its own line.

matz = { "First name" => "Yukihiro",

"Last name" => "Matsumoto",

"Age" => 47,

"Nationality" => "Japanese",

"Nickname" => "Matz"

}

matz.each do |key, value|

puts matz[key]

end

* **Nil: a Formal Introduction**

What happens if you **try to access a key that doesn't exist,** though?

In many languages, you'll get an error of some kind. Not so **in Ruby**: you'll **instead get the special value nil.**

Along with **false, nil is one of two non-true values in Ruby**. (**Every other object is regarded as "truthy,"** meaning that if you were to ***type if 2***or ***if "bacon"***, the code in **that if statement would be run.)**

It's important to realize that false and nil are **not** the same thing: **false means "not true,"** while**nil is Ruby's way of saying "nothing at all."**

you can access the value of a particular key in a hash with

hash\_name[key]

* **Setting Your Own Default**

You **don't have to settle for nil as a default value.**

If you **create your hash using the Hash.new** syntax, you can **specify a default like so:**

my\_hash = Hash.new("Trady Blix")

Now if you **try to access a nonexistent key in my\_hash**, you'll get "Trady Blix" as a result.

<http://ruby-doc.org/core-1.9.3/Hash.html>

no\_nil\_hash = Hash.new("default value")

to set ("default value") as the default value

**A Key of a Different Color**

We can certainly use strings as Ruby hash keys; as we've seen, there's always more than one way to do something in Ruby. However, the Rubyist's approach would be to use **symbols**.

Check out the code in the editor. Those funny-looking variables that start with colons are symbols. Click Next to the next section for a symbol rundown.

menagerie = {

:foxes => 2,

:giraffe => 1,

:weezards => 17,

:elves => 1,

:canaries => 4,

:ham => 1

}

* **What's a Symbol?**

You can think of a **Ruby symbol** as a **sort of name.** It's important to remember that ***symbols aren't strings***:

"string" == :string **# false**

there's a **key behavior of symbols that makes them different from strings**. While there can be **multiple different strings that all have the same value**, there's ***only one copy of any particular symbol at a given time***.

The .object\_id method gets the ID of an object—it's how Ruby knows whether two objects are the exact same object.

puts "string".object\_id

puts "string".object\_id

puts :symbol.object\_id

puts :symbol.object\_id

* **Symbol Syntax**

Symbols always **start with a colon (:).** They must be **valid Ruby variable names,** so the **first character after the colon** has to be a **letter or underscore (\_);** after that, any combination of letters, numbers, and underscores is allowed.

**don't put any spaces in your symbol name**

:my symbol

# Don't do this!

:my\_symbol

# Do this instead.

my\_first\_symbol = :greetings

* **What are Symbols Used For?**

Symbols pop up in a lot of places in Ruby, but they're **primarily used either as hash keys** or for **referencing method names**.

sounds = {

:cat => "meow",

:dog => "woof",

:computer => 10010110,

}

Symbols make **good hash keys** for a few reasons:

1. **They're immutable**, meaning they **can't be changed once they're created**;
2. **Only one copy of any symbol exists at a given time, so they save memory;**
3. **Symbol-as-keys** are faster than **strings-as-keys** because of the above two reasons.

symbol\_hash = {

:one => 1,

:two => 2, # Fill in these two blanks!

:three => 3,

}

* **Converting Between Symbols and Strings**

**Converting between strings and symbols is a snap.**

:sasquatch.to\_s

# ==> "sasquatch"

"sasquatch".to\_sym

# ==> :sasquatch

**The .to\_s and .to\_sym methods**

strings = ["HTML", "CSS", "JavaScript", "Python", "Ruby"]

# Add your code below!

symbols=[]

strings.each do |s|

new = s.to\_sym

symbols.push(new)

end

print symbols

* **Many Paths to the Same Summit**

**Besides using .to\_sym, you can also use*.intern.***

This will **internalize the string into a symbol** and *works just like .to\_sym:*

"hello".intern

# ==> :hello

When you're looking at someone else's code, **you might see .to\_sym or .intern (or both!) when converting strings to symbols**.

strings = ["HTML", "CSS", "JavaScript", "Python", "Ruby"]

# Add your code below!

symbols=[]

strings.each do |s|

new = s.intern

symbols.push(new)

end

print symbols

* **All Aboard the Hash Rocket!**

The **hash syntax** you've seen so far (with t**he => symbol between keys and values**) is sometimes nicknamed th*e****hash rocket****style.*

numbers = {

:one => 1,

:two => "two",

:three => 3,

}

Let's build a hash from the ground up using symbols as keys.

movies = {

:theMummy => "7/10, action",

:homeAlone => "6.5/10, family comedy",

}

* **The Hash Rocket Has Landed**

However, the hash **syntax changed in Ruby 1.9.**

new\_hash = {

one: 1,

two: 2,

three: 3

}

The two changes are:

1. You **put the colon at the end of the symbol**, not at the beginning;
2. You **don't need the hash rocket** anymore.

It's important to note that even though these ***keys have colons at the end instead of the beginning***, **they're still *symbols!***

puts new\_hash

# => { :one => 1, :two => 2, :three => 3 }

movies = {

theMummy: "7/10, action",

homeAlone: "6.5/10, family comedy"

}

* **Dare to Compare**

We mentioned that **hash lookup is faster with symbol keys than with string keys**. Here, we'll prove it!

The code in the editor uses some new syntax, so don't worry about understanding all of it just yet. It **builds two alphabet hashes**: one **that pairs string letters with their place in the alphabet** ( "a" with 1, "b" with 2...) and **one that uses symbols** (:a with 1, :b with 2...). We'll look up the letter "r" 100,000 times to see which process runs faster!

It's good to keep in mind that the numbers you'll see are only fractions of a second apart, and we did the hash lookup *100,000 times* each. It's not much of a performance increase to use symbols in this case, but it's definitely there!

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."

* **Becoming More Selective**

We know how to **grab a specific value from a hash** by **specifying the associated key,** but what if we **want to filter a hash for values** that **meet certain criteria**? we can use***.select.***

grades = {

alice: 100,

bob: 92,

chris: 95,

dave: 97

}

grades.select { |name, grade| grade < 97 }

# ==> { :bob => 92, :chris => 95 }

grades.select { |k, v| k == :alice }

# ==> { :alice => 100 }

1. In the example above, we first **create a grades hash** that **maps symbols to integers.**
2. Then we **call the .select method** and **pass in a block of code.** The block **contains an expression for selecting matching key/value pairs**. It ***returns a hash*** containing :bob and :chris.
3. Finally, we **call the .select method** again. Our block looks only for the key :alice. This is an inefficient method of getting a key/value pair, but it shows **that .select does not modify the hash.**

movie\_ratings = {

memento: 3,

primer: 3.5,

the\_matrix: 5,

truman\_show: 4,

red\_dawn: 1.5,

skyfall: 4,

alex\_cross: 2,

uhf: 1,

lion\_king: 3.5

}

# Add your code below!

good\_movies = movie\_ratings.select do |key, value|

value>3

end

* **More Methods, More Solutions**

We've often found we only want the key or value associated with a key/value pair, and it's kind of a pain to put both into our block and only work with one.

Can **we iterate over *just* keys or *just* values**?

Ruby includes two hash methods, .each\_key and .each\_value, that do exactly what you'd expect:

my\_hash = {

one: 1,

two: 2,

three: 3

}

my\_hash.each\_key { |k| print k, " " }

# ==> one two three

my\_hash.each\_value { |v| print v, " " }

# ==> 1 2 3

Let's wrap up our study of Ruby hashes and symbols by testing these methods out.

movie\_ratings = {

memento: 3,

primer: 3.5,

the\_matrix: 3,

truman\_show: 4,

red\_dawn: 1.5,

skyfall: 4,

alex\_cross: 2,

uhf: 1,

lion\_king: 3.5

}

# Add your code below!

movie\_ratings.each\_key do |key|

puts key

end

**A NIGHT AT THE MOVIES**

**What You'll Be Building**

Keeping track of all the parts of our digital lives is a pain. Now that you know how to write Ruby, however, you can make things easy for yourself! Let's start by creating a program that will keep track of our movie ratings.

It'll do one of four things: add a new movie to a hash, update the rating for an existing movie, display the movies and ratings that are already in the hash, or delete a movie from the hash. If it doesn't receive one of those four commands, the program will output some kind of error message.

This project will give you a lot of room for creativity, but we know sometimes it can be a little disorienting to not have strict instructions. If you ever feel lost, don't hesitate to check out the example code in this exercise to help you along!

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

**Setting Up**

let's **create a hash to hold our movies and their ratings**, and let's **prompt the user for input** so we can eventually **store movie/ratings pairs in our hash.**

favorite\_foods = {

'vegetable' => 'broccoli'

}

puts "Do you like coding in Ruby?"

answer = gets.chomp

1. A **hash is a way of storing data by a specifiable key**, as opposed to an **array which can only sort using numbers.** It is created like { } above.
2. puts asks a question on the command line, here we ask if you like coding in Ruby.
3. In order to get the user input, we have to call .chomp on gets

movies = {

Interstellar: 9,

theMummy: 6,

cloudAtlas: 9,

}

puts "What would you like to do today?"

choice = gets.chomp

* **The Case Statement**

**the case statement**, which will **decide what actions to take** based on **what the user types in.**

Ruby provides us with a concise alternative to if/if else: the case statement. The syntax looks like this:

case language

when "JS"

puts "Websites!"

when "Python"

puts "Science!"

when "Ruby"

puts "Web apps!"

else

puts "I don't know!"

end

**The else (default value)** is what the case statement will do **if it doesn't match any of its when statements to the case**(in this case, the value of language).

case variable\_to\_check

when value\_1

# Do something!

when value\_2

# Do something else!

when value\_3

# Do yet another thing!

when value\_4

# And so on and so forth

else

# Default thing to do

end

case choice

when "add"

puts "Added!"

when "update"

puts "Updated!"

when "display"

puts "Movies!"

when "delete"

puts "Deleted!"

else

puts "Error!"

end

**Prompting: Redux!**

Great! Let's build out each part of the case, one step at a time. We'll start with the "add" branch.

Inside your when "add" block, remove the puts "Added!" statement.

In its place, prompt the user for a movie title. Save the result in a new variable called title. (*Your code already has an example of how to do this!*)

Next, prompt the user for the rating of the movie. Save that in a new variable called rating.

Add that movie/rating pair to the movies hash and puts a message indicating the pair was added. (No need for to\_sym or to\_i just yet!)

Check the hint if you need help!

when "add"

puts "Movie to add: ? "

title = gets.chomp

puts "Movie rating: ? "

rating = gets.chomp

movies[title.to\_s] = rating

* **Not My Type**

Perfect! Our program is really coming along.

You might have wondered how we're going to get our movies and ratings from the user—**which come in as strings—into the hash where we want our movies to be symbols and our ratings to be integers**.

Ruby's **.to\_sym method** can convert **a string to a symbol**, and .**to\_i**will **convert a string to an integer.**

when "add"

puts "What movie would you like to add? "

title = gets.chomp.to\_sym

puts "What rating does the movie have? "

rating = gets.chomp.to\_i

movies[title] = rating

**Error! Error!**

All right! We're nearly done with the "add" part of our case. The final thing we'll want to do is perform a check to see whether the movie to be added is already in the hash.

To do this, we'll add an if/else statement.

The .nil? method will return true if the object it's called on is nil, and false otherwise:

nil\_variable = nil

age = 26

nil\_variable.nil?

# true

age.nil?

# false

when "add"

puts "Movie? "

title = gets.chomp.to\_sym

puts "Rating? "

rating = gets.chomp.to\_i

if movies[title].nil?

movies[title] = rating

else

puts "Movie exists in list"

end

* **Update**

Perfect! Let's move on to the next branch of our case statement, which handles updating an existing movie in the hash. (This should be very similar to the work we did in the "add" branch!) We'll do this in three steps:

Inside your when "update" block, remove the puts "Updated!" statement.

Prompt the user for a movie title. Store it in title.

if the movies[title] is nil, then the movie is not in the hash. Please puts a string telling the user of their error.

Otherwise (else), we need to update the movies hash. Prompt the user for a new rating. Set the movie's rating to that new value.

when "update"

puts "Movie? "

title = gets.chomp.to\_sym

if movies[title].nil?

puts "Movie not in list, yet"

else

puts "Rating? "

rating = gets.chomp.to\_i

movies[title] = rating

end

**Display**

Awesome! Now let's handle displaying the contents of our movies hash. This will be a little different from what we did for the "add" and "update" branches.

**Instructions**

**1.**

First, remove the puts "Movies!" when the user types "display".

Next, iterate through the hash using .eachand puts each movie/rating pair. The format should be "#{movie}: #{rating}. Make sure to test it out!

when "display"

movies.each do |title, rating|

puts "#{title}: #{rating}"

end

**Delete**

Almost there! Let's handle the "delete" part of our case statement, which will remove whatever key the user specifies from the hash. (This will be very similar to what we did for "add" and "update.")

**Ruby makes it easy to remove a movie/rating pair from our hash**: we just write **movies.delete(title)**

Go ahead and remove the puts "Deleted!"when the user types "delete".

Get the title from the user.

Include an if/else statement that puts an error if the movie's not in the hash; if it's there, use .delete to remove it as shown above.Make sure to test it out!

when "delete"

puts "Title? "

title = gets.chomp.to\_sym

if movies[title].nil?

puts "no such movie in list"

movies.delete(title)

end

**Nice Work!**

Fantastic! You built a little app with only a few dozen lines of code. Impressive, isn't it?

The four verbs your program knows—add, display, update, and delete—are universal. This acronym is better known as [CRUD](http://en.wikipedia.org/wiki/Create,_read,_update_and_delete)for **c**reate, **r**ead, **u**pdate, and **d**elete (respectively). These are the actions you take when you update an entry in a database, ask a website for information, or write a blog post. Being familiar with this setup is good, because you'll see it in everything from [API](http://en.wikipedia.org/wiki/Application_programming_interface)calls to web frameworks like Ruby on Rails.

* REFACTORING

**THE ZEN OF RUBY**

**Ruby is a Delight**

As a language, Ruby prioritizes programmer productivity over program optimization. This means that Ruby may not always run a program in the fastest way possible, but it strives to be a language that programmers (like you!) find easy and fun to use. The more intuitive a language's syntax is, the more productive its users can be. You're in control, not the machine!

Check out the code in the editor. It looks almost like English, doesn't it?

ruby\_is\_eloquent = true

ruby\_is\_ugly = false

puts "Ruby is eloquent!" if ruby\_is\_eloquent

puts "Ruby's not ugly!" unless ruby\_is\_ugly

**A Simpler 'If'**

If the "do something" is a short, simple expression, however, we can move it up into a single line (as you saw in the last exercise). The syntax looks like this:

expression if Boolean

Ruby will expect an **expression followed by if followed by a boolean**. **The order is important.** You can do this:

***puts "It's true!" if true***

but not this:

if true puts "It's true!"

It's also important to note that you **don't need an end when you write your if statement all on one line.**

puts "Good" if true

**The One-Line Unless**

You can do the exact same thing with the unless statement. The order is the same as before: something for Ruby to do, the unless keyword, and then an expression that evaluates to true or false.

Remember, you don't need an end when you write a one-line if or unless!

puts "Hello!" unless false

puts "Better" unless false

* **One Good Turn Deserves a Ternary**

An **even more concise version of if/else**is the **ternary conditional expression**.

It's called **"ternary**" because it **takes three arguments**: **a boolean**, an **expression to evaluate if the boolean is true**, and an **expression to evaluate if the boolean is false**.

The syntax looks like this:

boolean ? Do this if true: Do this if false

An example might be

puts 3 < 4 ? "3 is less than 4!" : "3 is not less than 4."

Remember: the order of arguments is important, and you don't need an end for this version of if/else.

puts false ? "Better" : "Worse"

puts true ? "True!" : "False!"

* **When and Then: The Case Statement**

Ruby provides us with a **concise alternative to if/else statements**: **the case statement**.

The syntax looks like this:

case language

when "JS"

puts "Websites!"

when "Python"

puts "Science!"

when "Ruby"

puts "Web apps!"

else

puts "I don't know!"

end

But you can fold it up like so:

case language

when "JS" then puts "Websites!"

when "Python" then puts "Science!"

when "Ruby" then puts "Web apps!"

else puts "I don't know!"

end

puts "Hello there!"

greeting = gets.chomp

# Add your case statement below!

case greeting

when "English" then puts "Hello!"

when "French" then puts "Bonjour!"

when "German" then puts "Guten Tag!"

when "Finnish" then puts "Haloo!"

else puts "I don't know that language!"

end

* **Conditional Assignment**

**use the = operator to assign a value to a variable**. But what if **we only want to assign a variable *if it hasn't already been assigned***?

For this, we can **use the conditional assignment operator**:  ***||=***. It's made up of the or **(||) logical operator** and the **normal = assignment operator**.

favorite\_book = nil

puts favorite\_book

favorite\_book ||= "Cat's Cradle"

puts favorite\_book

favorite\_book ||= "Why's (Poignant) Guide to Ruby"

puts favorite\_book

favorite\_book = "Why's (Poignant) Guide to Ruby"

puts favorite\_book

OUTPUT:

Cat's Cradle

Cat's Cradle

Why's (Poignant) Guide to Ruby

favorite\_language||="Espanol"

puts favorite\_language

* **Implicit Return**

What if we **don't put a return statement in our method definition**?

For instance, if you don't tell a JavaScript function exactly what to return, it'll return undefined. For Python, the default return value is None.

Ruby's ***methods will return the result of the last evaluated expression***.

This means that if you have a Ruby method like this one:

def add(a,b)

return a + b

end

You can simply write:

def add(a,b)

a + b

end

And either way, when you call add(1,1), you'll get 2.

def multiple\_of\_three(n)

n % 3 == 0 ? "True" : "False"

end

* **Short-Circuit Evaluation**

boolean operators **and** (&&) and **or** (||)

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.

Remember how **Ruby returns the result of the last expression it evaluated**?

def a

puts "A was evaluated!"

return true

end

def b

puts "B was also evaluated!"

return true

end

puts a || b

puts "------"

puts a && b

Check out the code in the editor, then click Run. Because **only false and nil are false values in Ruby**, both strings are treated as true. Ruby knows true || anything is true, so in a || b, it only evaluates a. Since it might encounter a false in the b part of a && b, however, it has to evaluate b, which we see in the result!

OUTPUT:

A was evaluated!

true

------

A was evaluated!

B was also evaluated!

true

* **The Right Tool for the Job**

perform a **repetitive task** in your programs. Many programming languages allow you to do this **with a for loop**, and while

If we want to **do something** **a specific number of times**, we can use **the .times method**, like so:

5.times { puts "Odelay!" }

# Prints 5 "Odelay!"s on separate lines

If we want to repeat an action for every element in a collection, we can use .each:

[1, 2, 3].each { |x| puts x \* 10 }

# Prints 10, 20, 30 on separate lines

my\_array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

my\_array.each { |x| puts x if x%2==0 }

my\_array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

my\_array.each { |x| puts x if x.even? }

* **Up the Down Staircase**

If we **know the range of numbers we'd like to include,** we can **use .upto and .downto**.

This is a much more Rubyist solution than trying to use a for loop that stops when a **counter variable hits a certain value.**

We might use .**upto to print out a specific range of values**:

95.upto(100) { |num| print num, " " }

# Prints 95 96 97 98 99 100

and we can use .**downto to do the same thing with descending values**.

Do you think**.upto and .downto work on the alphabet**

"A".upto("Z") { |letter| puts letter }

"L".upto("P") { |letter| puts letter }

* **Call and Response**

**symbols are awesome for referencing method names?**

Well***, .respond\_to?*takes a symbol** and **returns true if an object can receive that method and false otherwise**.

[1, 2, 3].respond\_to?(:push)

would return true, since you can call .push on an array object.

[1, 2, 3].respond\_to?(:to\_sym)

would return false, since you can't turn an array into a symbol.

age = 26

# Add your code below!

age.respond\_to?(:next)

* **Being Pushy**

Ruby has some nice **shortcuts for common method names.** : one is for **.push**

**Instead of typing out the .push method name**, you can simply use **<<**

the **concatenation operator** (also known as "**the shovel")** to **add an element to the end of an array**:

**[1, 2, 3] << 4**

# ==> [1, 2, 3, 4]

Good news: it also **works on strings!** You can do:

"**Yukihiro " << "Matsumoto**"

# ==> "Yukihiro Matsumoto"

alphabet = ["a", "b", "c"]

alphabet << "d" # Update me!

caption = "A giraffe surrounded by "

caption << "weezards!" # Me, too!

puts caption

* **String Interpolation**

You can always use plain old **+ or << to add a variable value into a string**:

drink = "espresso"

"I love " + drink

# ==> I love espresso

"I love " << drink

# ==> I love espresso

But if you want to do it for non-string values, you have to use .to\_s to make it a string:

age = 26

"I am " + age.to\_s + " years old."

# ==> "I am 26 years old."

"I am " << age.to\_s << " years old."

# ==> "I am 26 years old."

This is complicated, and complicated is not the Ruby way. A better way to do this is with **string interpolation**. The syntax looks like this:

"I love #{drink}."

# ==> I love espresso.

"I am #{age} years old."

# ==> I am 26 years old.

All you need to do is place the variable name inside #{} within a string!

favorite\_things = ["Ruby", "espresso", "candy"]

puts "A few of my favorite things:"

favorite\_things.each do |thing|

puts "I love #{thing}!"

end

* **One-Liners**

All right! Time to put your new knowledge to work by **refactoring** some existing code. **Refactoring** is just a fancy way of saying **we're improving the structure or appearance of our code without changing what it actually does.**

puts "One is less than two!" if 1 < 2

action if boolean

* **The Ternary Operator**

three = 3

puts three == 3 ? "Of course." : "What?"

# ==> puts "Of course."

puts 1 < 2 ? "One is less than two!" : "One is not less than two."

name = "Matz"

puts name == "Matz" ? "Hi Dad!" : "Who are you?"

* **In Case of Many Options**

Excellent. Regular if/else statements aren't the only ones we can refactor, though—a chain of if/elsif/else statements can clean up really nicely, too!

puts "What's your favorite language?"

language = gets.chomp

case language

when "Ruby" then puts "Ruby is great for web apps!"

when "Python" then puts "Python is great for science."

when "JavaScript" then puts "JavaScript makes websites awesome."

when "HTML" then puts "HTML is what websites are made of!"

when "CSS" then puts "CSS makes websites pretty."

else puts "I don't know that language!"

end

case variable when value1 then #Do something! when value2 then #Do something else! ... when value9 then #Do another thing! else # Do the default thing! end

* **Conditional Assignment**

Perfect! Now let's review conditional assignment. We'll take a break from strict editing mode and let you do a bit more writing.

favorite\_animal ||= "lion"

* **Implicit Return**

Next up: let's simplify our method madness by removing unnecessary returns from our code.

def square(number)

number\*\*2

end

**'For' Shame!**

All right! Last one: let's do something about the decidedly un-Ruby for loop in the editor.

10.times do

puts "Knock knock."

puts "Who's there?"

end

3.times do

puts "I'm a refactoring master!"

end

* **THE REFACTOR FACTORY**
* **What You'll Be Fixing**

The code in the editor is a Ruby method, first\_n\_primes, that takes a number *n* and generates a list of the first *n* prime numbers.

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

def first\_n\_primes(n)

return "n must be an integer." unless n.is\_a? Integer

return "n must be greater than 0." if n <= 0

Prime.first n

end

first\_n\_primes(10)

* **BLOCKS, PROCS, AND LAMBDAS**
* **You Know This!**

We said earlier that a **block is like a nameless method**, but that's not *quite* true. (**nameless methods, called *lambdas***)

A **Ruby block** is just a bit of **code that can be executed**.

Block syntax uses **either do..end or curly braces ({}),** like so:

[1, 2, 3].each do |num|

puts num

end

# ==> Prints 1, 2, 3 on separate lines

[1, 2, 3].each { |num|

puts num

}

# ==> Prints 1, 2, 3 on separate lines

**Blocks can be combined with methods like .each and .times** to ***execute an instruction*** **for each element** in a collection (**like a hash or array).**

* **Collect 'Em All**

Ruby methods that take blocks : ***collect***

The **collect method** **takes a block** and **applies the expression in the block to every element in an array.** Check it out:

my\_nums = [1, 2, 3]

my\_nums.collect { |num|

num \*\* 2

}

# ==> [1, 4, 9]

If we look at the **value of my\_nums, though, we'll see it hasn't changed:**

my\_nums # ==> [1, 2, 3]

This is because **.collect returns a *copy* of my\_nums**, but doesn't change (or **mutate**) the original my\_nums array.

If we want to do that, we can **use .collect! with an exclamation point:**

my\_nums.collect! { |num| num \*\* 2 }

# ==> [1, 4, 9]

my\_nums

# ==> [1, 4, 9]

Recall that the ! in Ruby means "this method could do something dangerous or unexpected!" In this case, it **mutates the original array instead of creating a new one.**

fibs = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

# Add your code below!

doubled\_fibs = fibs.collect { |number| number\*2 }

puts doubled\_fibs

* **Learning to Yield**

Why do some methods accept a block and others don't? It's because **methods that accept blocks** have **a way of transferring control from the calling method to the block and back again**. We can **build this into the methods we define by using the *yield* keyword.**

def block\_test

puts "We're in the method!"

puts "Yielding to the block..."

yield

puts "We're back in the method!"

end

block\_test { puts ">>> We're in the block!" }

OUTPUT:

We're in the method! Yielding to the block... >>> We're in the block! We're back in the method!

* **Yielding With Parameters**

You can also **pass parameters to yield**!

1. The **yield\_name method** is **defined with one parameter**, name.
2. On line 9, we call the yield\_name method and supply the argument **"Eric" for the name parameter**. Since **yield\_name has a yield statement**, we will also **need to supply a block.**
3. Inside the method, on line 2, we first puts an introductory statement.
4. Then we yield to the block and pass in "Kim".
5. In the block, n is now equal to "Kim" and we puts out "My name is Kim."
6. Back in the method, we puts out that we are in between the yields.
7. Then we yield to the block again. This time, we pass in "Eric" which we stored in the name parameter.
8. In the block, n is now equal to "Eric" and we puts out "My name is Eric."
9. Finally, we puts out a closing statement.

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!

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("Don") {|name| puts "My name is #{name}!"}

def double(num)

yield(num)

end

Define your own method, double, that accepts a single parameter and yields to a block. Then call it with a block that multiplies the number parameter by 2. You can double any number you like!

puts the result in order to see your yield in action!

double(16) {|x| puts x \* 2}

* **Keeping Your Code DRY**

**Blocks *are not* objects**, and this is one of the very few exceptions to the "everything is an object" rule in Ruby.

Because of this, **blocks can't be saved to variables**. For that, we'll need... **procs**!

You can think of a **proc as a "saved" block**: just like you can give a bit of code a name and turn it into a method, **you can name a block and turn it into a proc.**

Procs are great for keeping your code **DRY**, which stands for **D**on't **R**epeat **Y**ourself. With blocks, you have to write your code out each time you need it; **with a proc, you write your code once and can use it many times!**

multiples\_of\_3 = Proc.new do |n|

n % 3 == 0

end

print (1..100).to\_a.select(&multiples\_of\_3)

* **Proc Syntax**

**call Proc.new**and **pass in the block you want to save**.

Here's how we'd create a proc called cube that cubes a number:

cube = Proc.new { |x| x \*\* 3 }

We can then **pass the proc** **to a method** that **would otherwise take a block**,

[1, 2, 3].collect!(&cube)

# ==> [1, 8, 27]

[4, 5, 6].map!(&cube)

# ==> [64, 125, 216]

(The .collect! and .map! methods do the exact same thing.)

**The & is used to convert the cube proc into a block**

(since .collect! and .map! normally take a block). We'll do this any time **we pass a proc to a method that expects a block.**

The .**floor method rounds a float** (a number with a decimal) **down** to the **nearest integer**.

Write a proc called round\_down that will do this rounding (we've added the code to pass it to floats.collect).

floats = [1.2, 3.45, 0.91, 7.727, 11.42, 482.911]

# Write your code below this line!

round\_down = Proc.new {|x| x.floor }

# Write your code above this line!

ints = floats.collect(&round\_down)

print ints

* **Why Procs?**

Why bother saving our blocks as procs? There are two main advantages:

1. **Procs are full-fledged objects,** so they have all the powers and abilities of objects.
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.

over\_4\_feet = Proc.new do |height|

# Include rest of block code here

end

# Here at the amusement park, you have to be four feet tall

# or taller to ride the roller coaster. Let's use .select on

# each group to get only the ones four feet tall or taller.

group\_1 = [4.1, 5.5, 3.2, 3.3, 6.1, 3.9, 4.7]

group\_2 = [7.0, 3.8, 6.2, 6.1, 4.4, 4.9, 3.0]

group\_3 = [5.5, 5.1, 3.9, 4.3, 4.9, 3.2, 3.2]

# Complete this as a new Proc

over\_4\_feet = Proc.new {|height| height >= 4 }

# Change these three so that they use your new over\_4\_feet Proc

can\_ride\_1 = group\_1.select(&over\_4\_feet)

can\_ride\_2 = group\_2.select(&over\_4\_feet)

can\_ride\_3 = group\_3.select(&over\_4\_feet)

puts can\_ride\_1

puts can\_ride\_2

puts can\_ride\_3

* **Create Your Own!**

Okay! Time to take off the training wheels.

cube = Proc.new { |x| x \*\* 3 } [1, 2, 3].map(&cube) # [1, 8, 27]

You're going to create your very own method that calls your very own proc! We'll do this in two steps. Use the example above as a syntax reminder.

Create a method, greeter, that takes no arguments and yields to a block.

Create a Proc, phrase, that puts "Hello there!". Pass this to greeter instead of a block. (Don't forget to pass &phrase instead of just phrase!)

def greeter

yield

end

phrase = Proc.new { puts "Hello there!" }

greeter(&phrase)

**Call Me Maybe**

Nice work! Calling a proc with a method isn't too tricky. However, there's an even easier way.

Unlike blocks, we can call procs directly by using Ruby's .call method. Check it out!

test = Proc.new { # does something } test.call # does that something!

Remember: there's *always* more than one way to do something in Ruby.

hi = Proc.new { puts "Hello!"}

hi.call

* **Symbols, Meet Procs**

Well, you can ***also* convert symbols to procs** using that handy little **&.**

Check it out:

strings = ["1", "2", "3"]

nums = strings.map(&:to\_i)

# ==> [1, 2, 3]

By mapping &:to\_i over every element of strings, we turned each string into an integer!

Using the example in the instructions as a guide, use collect or map to create the strings\_array from the numbers\_array. Each element of strings\_array should be the string version of the corresponding element from the numbers\_array (that is, it should go ["1", "2", "3"... "10"]).

numbers\_array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

strings\_array = numbers\_array.collect(&:to\_s)

* **The Ruby Lambda**

**Like *procs, lambdas* are objects.** with the exception of a bit of syntax and a few behavioral quirks, lambdas are identical to procs.

Check out the code in the editor. See the lambda bit? Typing

lambda { puts "Hello!" }

Is just about the same as

Proc.new { puts "Hello!" }

In the example to the right, when ***we pass the lambda to lambda\_demo***, the me**thod calls the lambda and executes its code.**

def lambda\_demo(a\_lambda)

puts "I'm the method!"

a\_lambda.call

end

lambda\_demo(lambda { puts "I'm the lambda!" })

* **Lambda Syntax**

Lambdas are defined using the following syntax:

lambda { |param| block }

Lambdas are useful in the same situations in which you'd use a proc.

symbolize = lambda { |x| x.magic! }

strings = ["leonardo", "donatello", "raphael", "michaelangelo"]

# Write your code below this line!

symbolize = lambda { |param| param.to\_sym }

# Write your code above this line!

symbols = strings.collect(&symbolize)

print symbols

* **Lambdas vs. Procs**

two (2) main differences.

1. **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.**

1. **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.**

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.

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

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

See how the proc says Batman will win? This is because **proc 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!

**Now You Try!**

Great work! You've written your own lambda and seen how to pass it to a method. Now it's time for you to write a lambda *and* pass it to a method!

If you think this will be a lot like what you've already done with procs, you're exactly right. Just like with procs, we'll need to put & at the beginning of our lambda name when we pass it to the method, since this will convert the lambda into the block the method expects.

That symbolize lambda was pretty cool. Let's riff on it with a lambda that checks to see if each element in an array is a symbol. We can do this checking with the .is\_a? method, which returns true if an object is the type of object named and false otherwise:

:hello.is\_a? Symbol

# ==> true

The word **Symbol has to be capitalized** when you're **doing an .is\_a? check**

my\_array = ["raindrops", :kettles, "whiskers", :mittens, :packages]

# Add your code below!

symbol\_filter = lambda {|param| param.is\_a? Symbol}

symbols = my\_array.select(&symbol\_filter)

puts symbols

symbol\_filter = lambda { |x| x.magic! } my\_array.select(&lambda\_name)

* **Quick Review**

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**.

odds\_n\_ends = [:weezard, 42, "Trady Blix", 3, true, 19, 12.345]

ints = odds\_n\_ends.select { |element| element.is\_a? Integer}

puts ints

1. A **proc** is a **saved block** we can **use over and over**.

ages = [23, 101, 7, 104, 11, 94, 100, 121, 101, 70, 44]

# Add your code below!

under\_100 = Proc.new {|number| true if number<100 }

youngsters = ages.select(&under\_100)

puts youngsters

1. 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.**

crew = {

captain: "Picard",

first\_officer: "Riker",

lt\_cdr: "Data",

lt: "Worf",

ensign: "Ro",

counselor: "Troi",

chief\_engineer: "LaForge",

doctor: "Crusher"

}

# Add your code below!

first\_half = lambda {|key,value| value < "M" }

a\_to\_m = crew.select(&first\_half)

puts a\_to\_m

* **OBJECT-ORIENTED PROGRAMMING I**
* **Why Classes?**

Ruby is an **object-oriented programming language**, which means it **manipulates programming constructs called *objects***.

**Objects have methods**, which you've seen before, **and *attributes***, which are just data.

"Matz".length

# ==> 4

the "**Matz" object** is a **string** with a .**length method** and a **length attribute of 4.**

But **what exactly makes "Matz" a string**? The fact that it's an **instance of the String *class*.** A **class** is just a **way of organizing and producing objects with similar attributes and methods.**

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

* **Class Syntax**

A **basic class** consists only of the **class keyword** and the **name of the class**.

class NewClass

# Class magic here

end

**Our NewClass**has the ability to **create new Ruby objects** of **class NewClass**(just like "Hello!" is a String and 4 is a Fixnum).

By **convention, class names start with a capital letter** and use **CamelCase** instead of relying\_on\_underscores.

class Person

#class features

end

* **Classing It Up**

we **started our class definition** off **with a method *called initialize****.* It is **the function that "boots up" each object the class creates**.

class Person

def initialize

end

end

* **What's in a @name?**

we have to **make sure each person has a @name.**

In Ruby, we use **@** **before a variable to signify that it's an *instance variable***; **variable is attached to the *instance* of the class.**

class Car

def initialize(make, model)

@make = make

@model = model

end

end

kitt = Car.new("Pontiac", "Trans Am")

The code in the example above **creates an instance, kitt**, **of the class Car.**

**kitt** has his own **@make("Pontiac")** and **@model ("Trans Am")**. Those **variables belong to the kitt instance** : instance variables.

class Person

def initialize (name)

@name = name

end

end

* **Instantiating Your First Object**

We can **create an instance of a class** just by **calling .new**on the **class name**,

me = Person.new("Eric")

matz = Person.new("Yukihiro")

* **Scope it Out**

Another important **aspect of Ruby classes is *scope*** – **scope of a variable** is the **context in which it's visible to the program.**

When **dealing with classes**, you can have **variables that are available everywhere** (***global variables***), ones that are **only available inside certain methods (*local variables***), others that are **members of a certain class (*class variables*)**, and **variables that are only available to particular instances of a class (*instance variables*).**

The same goes for **methods:** some are **available everywhere**, some are only **available to members of a certain class**, and some **are only available to particular instance objects.**

variables start with $, @, or @@? This helps mark them as global, instance, and class variables (respectively).

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.

* Naming Your Variables

**instance variables begin with an @**

***Class variables***are *like instance variables*, but instead of belonging to an **instance of a class**, they **belong to the class itself.** Class variables always start with two @s:

@@files.

***Global variables***can be **declared in two ways:**

The first is one that's already familiar to you: you just **define the variable outside of any method or class**.

To **make a variable global from inside a method or class**, just **start it with a $**

$matz

class MyClass

$my\_variable = "Hello!"

end

puts $my\_variable

* **For Instance...**

**global variables** can be **changed from anywhere in your program**; **not a very good idea**; better to **create variables with limited scope**

For example, **instance variables belong to a particular object (or "instance of class ").**

class Person

def initialize(name, age, profession)

@name = name

@age = age

@profession = profession

end

end

* **Twice the @, Twice as Classy**

We can **create class variables** by **starting a variable name with two @ symbols**. **Class variables** are **attached to entire classes**:

class MyClass

@@class\_variable

end

Because there's **only one copy of a class variable** **shared by all instances of a class**.

**use a class variable** to **keep track of the number of instances of that class we've created**.

class Person

# Set your class variable to 0 on line 3

**@@people\_count = 0**

def initialize(name)

@name = name

# Increment your class variable on line 8

**@@people\_count += 1**

end

def self.number\_of\_instances

# Return your class variable on line 13

**return @@people\_count**

end

end

matz = Person.new("Yukihiro")

dhh = Person.new("David")

puts "Number of Person instances: #{Person.number\_of\_instances}"

* **Classes Are Serious Business**

Here we have a snippet of the [Rails source code](https://github.com/rails/rails" \t "_blank). See how they've created an instance of the RecordInvalid class?

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

* **Watch Your Step**

***Inheritance***is a tricky concept.

**Inheritance is the process** by which **one class takes on the attributes and methods of another**, and it's **used to express an *is-a* relationship**. For example, a **cartoon fox *is a* cartoon mammal**, so a **CartoonFox class could inherit from a CartoonMammal class.**

However, a Wizard is not an Elf, so it shouldn't inherit from the Elf class (even if they have a lot of magical attributes and methods in common). **Instead, both Wizard and Elf could ultimately inherit from the same MagicalBeing class.**

Check out the code in the editor. We've **defined a class, ApplicationError**, as well as a **SuperBadError class** that **inherits from ApplicationError**. Note that **we don't define the display\_error method in the body of SuperBadError**, but it will **still have access to that method via inheritance**.

class ApplicationError

def display\_error

puts "Error! Error!"

end

end

class SuperBadError < ApplicationError

end

err = SuperBadError.new

err.display\_error

* **Inheritance Syntax**

In Ruby, inheritance works like this:

class DerivedClass < BaseClass

# Some stuff!

end

**The derived class** is the **new class you're making** and **the base class is the class** from **which that new class inherits.** You can **read "<" as "inherits from**."

class Application

def initialize(name)

@name = name

end

end

# Add your code below!

class MyApp < Application

end

* **Override!**

Sometimes you'll want **one class that inherits from another** to **not only take on the methods and attributes of its parent**, but to ***override one or more of them.***

For instance, you might have an Email class that inherits from Message. Both classes might **have a send method** that sends them, but the e-mail version *may have to identify valid e-mail addresses and use a bunch of e-mail protocols*

that Message knows nothing about.

**Rather than add a send\_email method to your derived class** **and inherit a send method you'll never use**, you can instead just **explicitly create a send method in the Email class** and have it do all the email-sending work.

This **new version of send method will *override***(that is, replace) **the inherited version for any object that is an instance of Email.**

class Creature

def initialize(name)

@name = name

end

def fight

return "Punch to the chops!"

end

end

# Add your code below!

class Dragon < Creature

def fight

return "Breathes fire!"

end

end

* **When Good isn't Good Enough**

On the flip side, sometimes you'll be working **with a derived class (or *subclass***) and realize that **you've *overwritten* a method or attribute defined in that class' base class** (also called a *parent* or *superclass*) that you **actually need**.

Have no fear! You **can directly access the attributes or methods of a superclass** with Ruby's **built-in super keyword.**

The syntax looks like this:

class DerivedClass < Base

def some\_method

super(optional args)

# Some stuff

end

end

end

When you **call super from inside a method**, that tells Ruby to **look in the superclass of the current class** and **find a method with the same name** as the **one from which super is called**. If it finds it, Ruby **will use the superclass' version of the method.**

class Creature

def initialize(name)

@name = name

end

*def fight*

return "Punch to the chops!"

end

end

# Add your code below!

class Dragon < Creature

*def fight*

puts "Instead of breathing fire..."

**super**

end

end

* **There Can Be Only One!**

**Any given Ruby class** can have only **one superclass**.

Some languages allow a class to have more than one parent, which is a model called **multiple inheritance**.

However, there are **instances** where you **want to incorporate data or behavior from several classes into a single class**, and Ruby allows this through **the use of *mixins*.**

demonstrate what happens if you try to do multiple inheritance in Ruby.

The demo code we're about to show you includes a fancy trick: **if you want to end a Ruby statement without going to a new line**, you can **just type a semicolon.**

This means you can write something like

class Monkey

end

on just one line:

class Monkey; end.

This is a time saver when you're writing **something very short**, like **an empty class or method definition.**

class Creature

def initialize(name)

@name = name

end

end

class Person

def initialize(name)

@name = name

end

end

class Dragon < Creature; end

class Dragon < Person; end

get a superclass mismatch for class Dragon error if we try this.

* Review

class ClassName

def initialize(param1, param2)

@param1 = param1

@param2 = param2

end

end

obj = ClassName.new(parameters)

class Message

@@messages\_sent = 0

def initialize (from, to)

@from = from

@to = to

@@messages\_sent += 1

end

end

my\_message = Message.new("me", "you")

1.

class Email < Message

def initialize(subject)

@subject = subject

end

end

2.

class Email < Message

def initialize(from, to)

super

end

end

* **VIRTUAL COMPUTER**
* **What You'll Be Building**

Now that you've learned all about classes and objects in Ruby, you can create any kind of Ruby object your heart desires. In this project, we'll use our newfound **knowledge to create a class, Machine,** and **generate instances of that class that can manipulate imaginary files for us.**

class Machine

@@users = {}

def initialize(username, password)

@username = username

@password = password

@@users[username] = password

@files = {}

end

def create(filename)

time = Time.now

@files[filename] = time

puts "#{filename} was created by #{@username} at #{time}."

end

def Machine.get\_users

@@users

end

end

my\_machine = Machine.new("eric", 01234)

your\_machine = Machine.new("you", 56789)

my\_machine.create("groceries.txt")

your\_machine.create("todo.txt")

puts "Users: #{Machine.get\_users}"

declare a variable called time and set it equal to the current time using Time.now

* A class **method belongs to the class itself**, and for that reason, it's **prefixed with the class name.**

class Machine

def Machine.hello

puts "Hello from the machine!"

end

end

* Creating Class instance

my\_computer = Computer.new("user","pass")

class Computer

@@users = {}

def initialize (username, password)

@username = username

@password = password

@files = {}

@@users[username] = password

end

def create (filename)

time = Time.now

@files[filename] = time

puts "user:#{@username}, created the file #{filename} at #{time}"

end

def Computer.get\_users

return @@users

end

end

my\_computer = Computer.new("user","pass")

* **OBJECT-ORIENTED PROGRAMMING II**
* **Need-to-Know Basis**

But when **other people are working on or using your programs, they may attempt to muck around with the way different parts of your program** do their jobs.

For this reason, Ruby allows you to **explicitly make some methods public and others private.**

**Public methods** allow for an **interface** with **the rest of the program**; they say, "Hey! Ask me if you need to know something about my class or its instances."

**Private methods**, on the other hand, are for your classes to do their own work undisturbed.

when we try to call the PRIVATE bank\_account\_number method from outside the class!:

private method `bank\_account\_number' called for #<Context::Person:0x00000000e38578 @name="Eric", @age=26>

* **Going Public**

**Methods are public by default in Ruby**, so if you don't specify public or private, your methods will be public.

Unless we want to make it clear to people reading our code which methods are public.

We do this by **putting *public* before our method definitions**, like so:

class ClassName

# Some class stuff

public

def public\_method

# public\_method stuff

end

end

Note that **everything after the public keyword through the end of the class definition will now be public unless we say otherwise**.

* **Private! Keep Out!**

Just as we use public to announce our public methods, we **use *private* to specify our private ones**:

class ClassName

# Some class stuff

public

# Public methods go here

def public\_method; end

private

# Private methods go here

def private\_method; end

end

**private methods** are just that: they're **private to the object where they are**

**defined**.

This means **you can only call these methods from other code inside the object ;** the method cannot be called with an ***explicit receiver*.**

You've been using **receivers** all along—these are **the objects on which methods are called!** Whenever you **call** **object.method**, **object is the receiver** of the method.

In order to **access private information**, we have to **create public methods** that **know how to get it**. This **separates the private *implementation***from the **public *interface***.

class Dog

def initialize (name, breed)

@name = name

@breed = breed

end

public

def bark

puts "Woof!"

end

private

def id

@id\_number = 12345

end

end

* **attr\_reader, attr\_writer**

We saw in [the lesson on classes](https://www.codecademy.com/courses/ruby-beginner-en-MFiQ6/" \t "_blank) that **Ruby needs methods in order to access attributes.** For instance, if we want to ***access a @name instance variable***, we had to write something like

def name

@name

end

Well, no longer! We can **use attr\_reader to access a variable and attr\_writer to change it.**

class Person

attr\_reader :name

attr\_writer :name

def initialize(name)

@name = name

end

end

Ruby does something like this for us automatically:

def name

@name

end

def name=(value)

@name = value

end

Like magic, **we can read and write variables as we please**! We just **pass our instance variables (as symbols) to attr\_reader or attr\_writer**.

(That name= might look funny, but you're allowed to **put an = sign in a method name.** That's just a Ruby convention saying, **"hey, this method sets a value!")**

class Person

attr\_reader :name

attr\_writer :job

def initialize(name, job)

@name = name

@job = job

end

end

* **attr\_accessor**

If we want to **both read and write a particular variable**, there's an even shorter *shortcut than using attr\_reader and attr\_writer.*

We can use **attr\_accessor to make a variable readable *and* writeable** in one fell swoop.

* **What's a Module?**

You can think of a **module** as a **toolbox that contains a set methods and constants.**

For that reason, **we keep a bunch of them in modules** and only **pull in those module toolboxes** when we **need the constants and methods inside**!

You can think of **modules as being very much like classes**, only ***modules can't create instances and can't have subclasses***. They're just **used to store things!**

module Circle

PI = 3.141592653589793

def Circle.area(radius)

PI \* radius\*\*2

end

def Circle.circumference(radius)

2 \* PI \* radius

end

end

* **Module Syntax**

You can **pull in pre-existing modules**, but you can **also make your own**.

**use the module keyword**, like so:

module ModuleName

# Bits 'n pieces

end

Like **class names, module names** are written in **CapitalizedCamelCase**.

It **doesn't make sense to include variables in modules**, since variables (by definition) change (or vary).

**Constants**, however, are supposed **to always stay the same**, so including helpful **constants in modules is a great idea.**

Ruby **doesn't *make* you keep the same value for a constant** once it's **initialized**, but **it will warn you if you try to change it**. Ruby **constants are written in ALL\_CAPS** and are **separated with underscores**

An example of a Ruby constant is **PI, which lives in the Math module** and is approximately equal to 3.141592653589793. We **created our own PI**in the previous exercise, but don't worry: **because they're in separate modules, Ruby knows to keep them separate.**

module MyLibrary

FAVE\_BOOK = "Les Miserables"

end

* **Resolve to Keep Learning**

One of the **main purposes of modules** is to **separate methods and constants into named spaces; namespacing**, and it's how Ruby doesn't confuse **Math::PI and Circle::PI.**

See that **double colon** - called the ***scope resolution operator,*** which is a fancy way of saying it tells Ruby ***where* you're looking for a specific bit of code**. If we say Math::PI, Ruby knows to look inside the Math module to get that PI, not any other PI (such as the one we created in Circle).

puts Math::PI

* **A Few Requirements**

Some modules, like **Math, are already present in the interpreter**.

**Others** need to **be explicitly brought in**, however, and we can do this **using require .**

We can do this simply by typing

require 'module'

We want to use the Ruby **Date module** to **show today's date**, but we haven't required it yet!

require 'date'

puts Date.today

* **Feeling Included**

We can also include a module.

**Any class that includes a certain module** can **use those module's methods**!

A nice effect of this is that you **no longer have to prepend your constants** and **methods with the module name**.

**simply write PI instead of Math::PI.**

class Angle

***include Math***

attr\_accessor :radians

def initialize(radians)

@radians = radians

end

def cosine

cos(@radians)

end

end

acute = Angle.new(1)

acute.cosine

* **The Marriage of Modules and Classes**

**mixed together the behaviors of a class and a module!**

When a **module is used to mix additional behavior and information into a class**, it's called a ***mixin*** ; allow us to **customize a class without having to rewrite code**

module Action

def jump

@distance = rand(4) + 2

puts "I jumped forward #{@distance} feet!"

end

end

class Rabbit

include Action

attr\_reader :name

def initialize(name)

@name = name

end

end

class Cricket

include Action

attr\_reader :name

def initialize(name)

@name = name

end

end

peter = Rabbit.new("Peter")

jiminy = Cricket.new("Jiminy")

peter.jump

jiminy.jump

* **Imitating Multiple Inheritance**

Now you understand why we said **mixins could give us the ability to mimic inheriting from more than one class**: by mixing in **traits from various modules** as needed, we can **add any combination of behaviors to our classes** we like!

# Create your module here!

module MartialArts

def swordsman

puts "I'm a swordsman."

end

end

class Ninja

include MartialArts

def initialize(clan)

@clan = clan

end

end

class Samurai

include MartialArts

def initialize(shogun)

@shogun = shogun

end

end

* **Extend Your Knowledge**

Whereas**include**mixes **a module's methods in at the instance level** (allowing **instances of a particular class** to use the methods), the **extend keyword mixes a module's methods at the *class* level**. This means that ***class itself***can **use the methods, as opposed to *instances* of the class**.

# ThePresent has a .now method that we'll extend to TheHereAnd

module ThePresent

def now

puts "It's #{Time.new.hour > 12 ? Time.new.hour - 12 : Time.new.hour}:#{Time.new.min} #{Time.new.hour > 12 ? 'PM' : 'AM'} (GMT)."

end

end

class TheHereAnd

extend ThePresent

end

TheHereAnd.now

class Application

attr\_accessor :status

def initialize; end

# Add your method here!

public

def print\_status

puts "All systems go!"

end

private

def password

return 12345

end

end

module Languages

FAVE = "Ruby"

end

class Master

include Languages

def initialize; end

def victory

puts FAVE

end

end

total = Master.new

total.victory

**BANKING ON RUBY**

**What You'll Be Building**

All right! Now that you know how to control the level of privacy in a Ruby class, we can use that to our advantage when creating objects. In this case, we'll be making an Account object with public methods to display balances and transfer funds, but which rely on private methods to make sure the user's PIN (personal identification number) is correct before approving transactions.

**Note**: We're just using banking as an example because it's a nice real-world analogy. This isn't for real banking, so don't use any real banking information!

class Account

attr\_reader :name, :balance

def initialize(name, balance=100)

@name = name

@balance = balance

end

def display\_balance(pin\_number)

puts pin\_number == pin ? "Balance: $#{@balance}." : pin\_error

end

def withdraw(pin\_number, amount)

if pin\_number == pin

@balance -= amount

puts "Withdrew #{amount}. New balance: $#{@balance}."

else

puts pin\_error

end

end

private

def pin

@pin = 1234

end

def pin\_error

"Access denied: incorrect PIN."

end

end

my\_account = Account.new("Eric", 1\_000\_000)

my\_account.withdraw(11, 500\_000)

my\_account.display\_balance(1234)

my\_account.withdraw(1234, 500\_000)

my\_account.display\_balance(1234)

We'll start off by creating our Account class. First, though, you probably noticed this bit of fanciness in the last exercise:

def initialize(name, balance=100)

@name = name

@balance = balance

What's that balance=100 doing? It's signifying an **optional parameter**. Ruby is saying **that you can pass one *or two* arguments to initialize**; if **you pass two**, it uses **your balance argument** to **set @balance**;

if you **only pass a name**, balance **gets a default value of 100**, and that's what gets **stored in @balance**.

You probably also noticed we **used underscores in our 1\_000\_000 (one million**). Ruby allows this, and it **makes it easier to read big numbers**!

class Account

attr\_reader :name

attr\_reader :balance

def initialize(name, balance=100)

@name = name

@balance = balance

end

public

def display\_balance(pin\_number)

if pin\_number == @pin

puts "Balance: $#{@balance}."

else

puts pin\_error

end

end

def withdraw (pin\_number, amount)

if pin\_number == pin

@balance -= amount

puts "Withdrew #{amount}. New balance: $#{@balance}."

else

puts pin\_error

end

end

private

def pin

@pin = 1234

end

def pin\_error

return "Access denied: incorrect PIN."

end

end

checking\_account = Account.new("Don",700)