Collections

Collections

Array:

```
> fruits = ["kiwi", "strawberry", "plum"]
=> ["kiwi", "strawberry", "plum"]
```

`fruits[0] = "kiwi", fruits[1] = "strawberry", etc.`

Associative array ("**hash**"):

```
> states = {"VA" => "Virginia", "MD" => "Maryland"}
=> {"VA" => "Virginia", "MD" => "Maryland"}
```

`states["VA"] = "Virginia", states["MD"] = "Maryland"`

Loops & Iterators: repeating yourself

```
> fruits[0]
kiwi
=>nil

> puts fruits[1]
strawberry
=> nil

> puts fruits[2]
plum
=> nil
```

this isn't fun or efficient!

.each: Do something repeatedly

```
> fruits.each do |fruit|
>     puts fruit
> end
kiwi
strawberry
plum
=> ["kiwi", "strawberry", "plum"]
```

Exercise

- Create an array of four places you would like to visit
- Print out each of those places **using a loop**

Example:

```
"I would like to visit Paris"
"I would like to visit Barcelona"
"I would like to visit Lima"
"I would like to visit Havana"
```

Conditional: do something if a condition is true

```
> fruits.each do |fruit|
>     puts fruit if fruit == "plum"
> end
plum
⇒ ["kiwi", "strawberry", "plum”
```

Exercise

- Create an array named `hilt_class` that contains the name of the people next to you. Be sure to include your own name.
- Using your `hilt_class` array, create a conditional that prints "`My name is (your name)`" for your name only

.each: for hashes

```
states = {"VA" => "Virginia", "MD" =>
"Maryland"}

states.each do |code, state|
    puts code.to_s + “is the code for " +
state.to_s
end
```

Power Tip:
There's a "short" code for placing variables within double-quoted (") strings:

```
puts "#{code} is the code for #{state}"
```

.times

```
counter = 0

10.times do
      counter = counter + 1
      puts "I'm at number " + counter.to_s
end
```

Fun with Arrays

```
array.sort      # sort the keys in an array
array.inspect   # quickly show items in an array
array.length    # returns the length (number of items)
array.empty?    # is the array empty?
array.reverse   # reverses the order of an array
array.uniq      # prints unique values in an array
```

There are **many** more methods! See the Array documentation.

Branching: Do something only under certain circumstances

```
if fruits[0] == "plum"
  puts fruits[0]
end
```

Remember: `=` is assignment, `==` is equivalence

Ruby one-liner:

```
puts fruits[0]if fruits[0] == "plum"
```

Branching: Do something only under certain circumstances

Use **`else`** and **`elsif`** for compound conditions.
Remember the "and" (&&) and "or" (||) operators?

```
if fruits[0] == "apple"
  puts "Yum!"
elsif fruits[0] == "cardboard" || fruits[0] == "sand"
  puts "Yuck!"
else
  puts "Not bad."
end
```

Branching

```
puts "Yuck" unless fruits[0] == "apple"

age = 5
case age
when 0..2
  puts "baby"
when 3..18
  puts "child"
else
  puts adult
end
```

while loop

continues while a condition is true
like an *each* loop with an *if*

```
counter = 0

while counter < 5
      puts counter
      counter += 1
end
```

Remember: `+=` **is shorthand for** `counter = counter + 1`

until loop

continues until a condition is met
like an *each* loop with an *if*

```
counter = 0

until counter == 5
      puts counter
      counter = counter + 1
end
```

Remember: = is assignment, == is equivalence

Splat Operator

```
numbers = (1..10).to_a
letters = ('a'..'z')

puts numbers.include?(5)
puts numbers.min
puts numbers.max

puts letters === 'c'
```

Interpreter

- Ruby is an **interpreted language**
  - Its code cannot be run directly
  - It must be run through a Ruby interpreter
- Most common interpreter is Matz's Ruby Interpreter (MRI)
  - There are others (jruby, rubinius, etc.)
- There are different ways to run code through a Ruby interpretor
  - Just used IRB
  - Now we'll use a file

Running code

- Why use a file? What's different from irb?
- **Note**: the directory your terminal is currently in is your ***working directory***

Code

Create a new file named `my_program.rb` in your working directory with this code

```
class Sample
  def hello
    puts "Hello World!"
  end
end

s = Sample.new
s.hello
```

Running Code

Run the save code in the terminal

```
$ ruby my_program.rb
Hello World!
```

Your Own Command Line Program

Hello World

Create a file named `hello.rb` and add the following code

```
puts "Hello, World!"
```

Now run it

```
$ ruby hello.rb
```

Arguments

Update `hello.rb` with the following

```
puts "Hello, #{ARGV.first}!"
```

Now run it with the following

```
$ ruby hello.rb Wayne
Hello, Wayne!
```

Conditionals

Refactor `hello.rb`

```
if ARGV.empty?
    puts "Hello, World!”
else
    puts "Hello, #{ARGV.first}!”
end
```

Now run it:

```
$ ruby hello.rb
Hello, World!
$ ruby hello.rb Wayne
Hello, Wayne!
```

Libraries

Useful behavior beyond the "basics"

- Ruby Standard Library
  - Files (CSV, text, etc)
  - Advanced math (linear algebra, encryption)
  - Internet (http, ftp, mail, etc.)
  - Documentation (rdoc)
- Ruby Gems
  - Just about everything else

Organization: code reuse

- Methods
  - Name code (like variables that name strings and numbers)
  - Take arguments
  - "Mini-scripts" || "Tiny commands"
  - Allows for code reuse

```
def add(x, y)
    puts x + y
end
```

Exercise

Create a **collection** of these authors and the year they kicked the bucket; print the collection in the following format:

Charles Dickens kicked the bucket in 1870.

Charles Dickens, 1870
William Thackeray, 1863
Anthony Trollope, 1882
Gerard Manley Hopkins, 1889

An Answer

```
authors = {
    "Charles Dickens" => "1870",
    "William Thackeray" => "1863",
    "Anthony Trollope" => "1882",
    "Gerard Manley Hopkins" => "1889"
}

authors.each do |author, year|
  puts author.to_s + " kicked the bucket in " + year.to_s
end
```

Can you write this as a method?

Exercise

A time traveller has suddenly appeared in the classroom!

Create a variable representing the traveller's year of origin (e.g., `year = 2000`) and greet our strange visitor with a different message if he is from the distant past (before 1900), the present era (1900-2020) or from the far future (beyond 2020).

An Answer

```
year = 2000

if year < 1900
  puts "Tell me of the past!"
elsif year >= 1900 && year <= 2020
  puts "I wish you were from a cooler era."
else
  puts "Hello, future traveller."
end
```

Rewrite (refactor) as a method to test different years

An Answer: improved

```
def greeting(year)
  if year < 1900
    puts "Tell me of the past!"
  elsif year >= 1900 && year <= 2020
    puts "I wish you were from a cooler era."
  else
    puts "Hello, future traveller."
  end
end
```

greeting 1878
greeting 2013
greeting 3000

Exercise

Create a collection of 19th- and 20th-century authors (or historical/political figures if that's your bag!) and their birth dates (historical accuracy doesn't matter). An example:

```
birth_dates = {"Wallace Stevens" => 1879}
```

Count the number of 19th-century birth dates and the number of 20th-century birth dates, then print the results like:

```
There are 3 19th-c. births and 2 20th-c. births in
my collection.
```

An Answer

```
birth_dates = {'Wallace Stevens' => 1897, 'Wayne
Graham' => 1977}

nineteenth_count = 0
twentieth_count = 0

birth_dates.each do |person, b_date|
  if b_date < 1900
    nineteenth_count += 1
  else
    twentieth_count += 1
  end
end
```

An Answer: continued

```
puts "There are " + nineteenth_count.to_s +
" 19th-c. births and " +
twentieth_count.to_s + " 20th-c. births in
my collection."
```

How might you expand this to capture additional centuries? Decades?

Questions?