Objectives

- Working with Files
- Metaprogramming in Ruby

Working with Files

Introduction

File handling is a fundamental part of programming. In Ruby, working with files allows us to store data permanently, read configuration files, save logs, or even build entire applications that depend on persistent storage.
Ruby provides a simple and powerful way to handle files using its built-in `File` and `IO` classes. These classes let you create, read, write, and modify files with ease.
There are three main operations we'll focus on:

- **Reading** from files
- **Writing** to files
- **Managing** files (like deleting or renaming)

Opening Files

To interact with a file, we first need to **open** it. Ruby uses the `File.open` method, which can be provided with different **modes** to specify how we want to interact with the file.

Common File Modes:

- `"r"` Read-only (default). The file must exist.
- `"w"` Write-only. Creates a new file or truncates an existing one.
- `"a"` Append. Data will be written at the end of the file.
- `"r+"` Read and write.
- `"w+"` Read and write, truncating the file first.
- `"a+"` Read and write, appending to the file.

Example: Opening a file for reading

```
file = File.open("example.txt", "r")
puts file.read
file.close
```

Always close the file after using it to free up system resources.

Reading Files

We have many ways to read data from file:

- `read` : Reads the entire file.
- `readline` : Reads one line at a time.
- `readlines` : Reads all lines into an array.

Example: Reading line by line

```
file = File.open("example.txt", "r")
file.each_line do |line|
  puts line
end
file.close
```

Shortcut using a block (auto-closes the file):

```
File.open("example.txt", "r") do |file|
  file.each_line { |line| puts line }
end
```

Writing to Files

To write to a file, we need first to open it in write (`"w"`) or append (`"a"`) mode. If the file doesn't exist, Ruby creates it automatically.
Example: Overwriting a file

```
File.open("output.txt", "w") do |file|
  file.puts "Hello, world!"
end
```

Example: Appending to a file

```
File.open("output.txt", "a") do |file|
  file.puts "Another line"
end
```

`puts` adds a newline, while `write` does not.

File Existence and Metadata

Ruby lets us check file properties before interacting with them using the `File` class methods.

```
puts File.exist?("output.txt")       # => true or false
puts File.size("output.txt")         # File size in bytes
puts File.directory?("my_folder")    # => true if it's a directory
puts File.file?("output.txt")        # => true if it's a file
```

Deleting and Renaming Files

Ruby also allows us to **rename and delete files easily** using `File.rename` and `File.delete`.

```
File.rename("old.txt", "new.txt")   # Rename a file
File.delete("unwanted.txt")         # Delete a file
```

Metaprogramming in Ruby

Introduction

Metaprogramming is one of Ruby's most powerful and expressive features. It allows us to **write code that writes code**, meaning our Ruby programs can dynamically create methods, define classes, or even alter behavior while running.

Dynamic Method Creation

Ruby lets us define methods at runtime using metaprogramming techniques.
Example: Using `define_method`

```
class Person
  [:name, :age, :city].each do |attribute|
    define_method(attribute) do
      instance_variable_get("@#{attribute}")
    end

    define_method("#{attribute}=") do |value|
      instance_variable_set("@#{attribute}", value)
    end
```

```
  end
end

p = Person.new
p.name = "Alice"
puts p.name  # => "Alice"
```

`method_missing` and `respond_to_missing?`

When we call a method that doesn't exist, Ruby raises an error. But with metaprogramming, we can intercept that behavior using `method_missing`.

Example: Dynamic attribute handler

```
class DynamicPerson
  def initialize
    @attributes = {}
  end

  def method_missing(name, *args)
    if name[-1] == "="
      @attributes[name.chop.to_sym] = args.first
    else
      @attributes[name]
    end
  end

  def respond_to_missing?(method_name, include_private = false)
    true
  end
end

dp = DynamicPerson.new
dp.name = "Bob"
puts dp.name  # => "Bob"
```

Defining Methods with `define_singleton_method`

We can also define methods on **individual objects** instead of the entire class.
Example: Singleton method

```
dog = "Barky"

def dog.speak
  "Woof!"
end

puts dog.speak  # => "Woof!"
```

Class and Instance Introspection

Ruby allows us to inspect and manipulate classes and objects at runtime. This is especially useful for debugging or dynamically interacting with unknown objects.
Examples:

```
puts "Hello".class                  # => String
puts 5.instance_of?(Integer)       # => true
puts "text".respond_to?(:upcase)   # => true
puts Person.instance_methods(false)
```

Opening and Modifying Existing Classes

In Ruby, we're free to reopen any class even built-in ones and add or modify methods.
Example: Add a method to `String`

```
class String
  def shout
    self.upcase + "!"
  end
end

puts "hello".shout  # => "HELLO!"
```

Final Project

Using everything we've learned during this course, create a Ruby project that solves a problem you face in your daily life.
Think about tasks you often repeat, information you want to organize, or anything you wish could be automated then build a solution using Ruby!