Objectives

- Introduction to Programming Languages
- Ruby Programming
- Variables and Data types
- User Input and Output

Introduction to Programming Languages

Programming languages are tools developed by computer scientists to facilitate communication with machines. Rather than writing complex instructions in binary (which quickly becomes unmanageable for large programs), we use programming languages with more human-readable syntax. These languages make it easier to write, understand, and maintain code.

Although these high-level languages are easier for humans to use, computers cannot directly understand them. Instead of a compiler, which is commonly used in languages like C, many modern languages like Ruby use an **interpreter**. An interpreter reads the code line by line and executes it directly, without converting the entire program into machine code first. This allows for quick testing and flexibility during development.

Ruby Programming

Introduction

Ruby is a dynamic, high-level programming language known for its simplicity and elegance. It was designed to be intuitive and developer-friendly, emphasizing readability and productivity. Ruby powers many applications, and it's especially well known for its use in web development through the **Ruby on Rails** framework.
One of Ruby's core strengths is its **object-oriented** nature in Ruby, *everything is an object*, including numbers and strings. It also supports functional and procedural programming styles, offering flexibility for different types of projects.

Ruby is **interpreted**, meaning there's no need to compile your code. You simply write the code and run it immediately, which makes it great for rapid development and prototyping. Additionally, Ruby code is **portable**: it can run on different systems without needing to be recompiled.

Installing Ruby

To start writing and running Ruby programs on our PC, we can follow these steps:

On Windows:

1. Go to the official Ruby installer for Windows: https://rubyinstaller.org
2. Download the latest version (with DevKit if available).
3. Run the installer and check the box that says **"Add Ruby executables to your PATH"**.
4. Open a terminal (Command Prompt or PowerShell), and type `ruby -v` to confirm Ruby is installed.

On macOS:

macOS comes with Ruby pre-installed, but we can install the latest version using Homebrew:

```
brew install ruby
```

After the installation, we might need to add Ruby to our path manually. We can check if it worked by typing `ruby -v` in the terminal.

On Linux:

We can use the system's package manager to install Ruby:

```
sudo apt install ruby   # for Ubuntu/Debian
sudo dnf install ruby   # for Fedora
```

Then we check the version using `ruby -v`.

Text Editor vs IDE in Ruby Development

A **text editor** is a basic program that allows us to write and format text. You can use any text editor (like VS Code, Sublime Text, or even Notepad) to write Ruby code. To execute Ruby programs, you just need to have Ruby installed on your system — no separate compiler is required. You can run Ruby code from the command line using the `ruby` command.

An **IDE (Integrated Development Environment)** is a more feature-rich environment that often includes a text editor, debugger, file manager, and integrated terminal. For Ruby, some popular IDEs and editors with Ruby support include **RubyMine**, **Visual Studio Code**, and **Aptana Studio**. These tools often offer syntax highlighting, code completion, and error checking, which make development more efficient.

Running Our First Ruby Program

Let’s write and run our very first Ruby program the classic **"Hello, World!"**.

Create the Ruby File

We start by opening a text editor or IDE. Then we type the following code:

```
puts "Hello, world!"
```

Save the File

We save the file with a `.rb` extension

```
hello.rb
```

The `.rb` tells our system that this is a Ruby script file.

Run the Program

To run our program, we open the terminal or command prompt, navigate to the folder where we saved the file, and type:

```
ruby hello.rb
```

If everything is set up correctly, we’ll see the message:

```
Hello, world!
```

Variables and Data Types

Variables

Variables serve as the fundamental building blocks in our programs - they're like labeled storage containers that hold our data. These containers can store various types of information including numbers, text (strings), true/false values (booleans), and more complex data structures.
Ruby is **Dynamic Typing** language it automatically determines the variable type at runtime, so we don't need to explicitly declare types. This gives us flexibility to assign different data types to the same variable.

Creating Variables

To create a variable in Ruby, we start by choosing a **meaningful name**. Variable names should:

- Begin with a lowercase letter or an underscore (`_`)
- Contain only letters, numbers, or underscores
  We assign a value to a variable using the `=` symbol.

```
name = "Alice"
age = 25
greeting_message = "Hello, #{name}!"
```

Data Types

Ruby supports several built-in data types that allow us to store, manipulate, and interact with different kinds of information. Each data type is treated as an object in Ruby, making the language both flexible and powerful.

Numbers

Ruby supports both **integers** and **floating-point numbers**. These are used for mathematical calculations, counting, measuring, and any numeric operations.

```
age = 30         # Integer
price = 19.99    # Float
```

We can perform arithmetic operations using the following operators:

- `+` (addition)
- `-` (subtraction)
- `*` (multiplication)
- `/` (division)
- `%` (modulus or remainder)

```
total = 10 + 5
half = 20 / 2.0
```

Strings

Strings are sequences of characters enclosed in either **double** (`"`) or **single** (`'`) quotes. We use them to store text such as names, sentences, or messages.

```
name = "Alice"
```

```
greeting = 'Hello, world!'
```

We can combine (concatenate) strings using the `+` operator, or insert values inside strings using **interpolation**:

```
first_name = "Alice"
last_name = "Joly"
greet = "Hello, #{first_name + " " + last_name}!"
```

Strings in Ruby are full-featured objects, and we can call many useful methods on them, like `.length`, `.upcase`, `.downcase`, and `.include?`.

Booleans

Boolean data types represent **logical values**: `true` or `false`. They're commonly used in conditions, comparisons, and control structures.

```
is_logged_in = true
has_permission = false
```

Arrays

Arrays are ordered collections that can store multiple values in a single variable. Each value has an **index**, starting at 0.

```
colors = ["red", "green", "blue"]
puts colors[0]    # => "red"
```

Ruby arrays can hold values of **any type**, including other arrays or hashes:

```
mixed = [1, "hello", true]
```

We can use methods like `.push`, `.pop`, `.length`, and `.each` to manipulate arrays.

Hashes

Hashes store **key-value pairs**, similar to dictionaries or maps in other languages. They're great for representing structured data.

```
person = {
  "name" => "Alice",
  "age" => 30
}
puts person["name"]  # => "Alice"
```

Ruby also supports **symbols** as hash keys, which are more memory-efficient:

```
person = {
  name: "Alice",
  age: 30
}
```

Symbols

Symbols are identifiers that look like strings but are **immutable** and **memory-efficient**. We often use them as keys in hashes or for naming things.

```
:username
:email
```

They are especially useful when we need identifiers that don't need to change.

Nil

The special value `nil` represents **"nothing"** or **"no value"** in Ruby. It's Ruby's version of `null` or `None`.

```
middle_name = nil
```

`nil` is commonly used to indicate the absence of a value, such as when a method doesn't return anything or a variable hasn't been assigned yet.

Constant

Constants are variables meant to hold values that should not change during program execution. They are declared by starting the variable name with an uppercase letter, though by convention they are typically written in ALL_CAPS.
Ruby allows reassignment to constants but issues a warning, This is different from many other languages where constants are truly immutable

```
PI = 3.14159
MAX_USERS = 100
DEFAULT_TIMEOUT = 30
```

Conveting between Types

Convert to Integer

We use `.to_i` to convert a value to an **integer**:

```
"42".to_i      # => 42
"hello".to_i   # => 0 (non-numeric strings become 0)
```

Convert to Float

We use `.to_f` to convert a value to a **floating-point number**:

```
"3.14".to_f    # => 3.14
"10".to_f      # => 10.0
```

Convert to String

Use `.to_s` to convert a number or boolean into a **string**:

```
100.to_s       # => "100"
true.to_s      # => "true"
```

Other Methods

Method	Converts To	Example
`.to_i`	Integer	`"50".to_i` → `50`
`.to_f`	Float	`"3.5".to_f` → `3.5`
`.to_s`	String	`false.to_s` → `"false"`
`.to_a`	Array (if possible)	Used on ranges or hash values
`.to_sym`	Symbol	`"name".to_sym` → `:name`

Comments

Comments are notes we write in our code to explain what it does. They are **ignored by Ruby** when the program runs they're just for us (and other humans) to understand the code better.

Single-line Comments

We use the `#` symbol for single-line comments. Anything after the `#` on that line is treated as a comment.

```
# This is a single-line comment
puts "Hello, world!"  # This prints a greeting to the screen
```

Multi-line Comments

While Ruby doesn’t have a built-in multi-line comment symbol like some other languages, we can write multiple lines with `#` at the start of each:

```
# This is a multi-line comment.
# It spans several lines.
# Each line starts with a hash symbol.
```

Alternatively, Ruby supports `=begin` and `=end` for block comments

```
=begin
This is also a multi-line comment.
It starts with =begin and ends with =end.
Useful for larger comment blocks.
=end
```

User Input and Output

In most programs, interacting with the user is essential. Ruby makes it easy to display information (output) and receive input from the user.

Output

We use the `puts` and `print` methods to show text on the screen.

- `puts` adds a **newline** after the output.
- `print` keeps the cursor on the **same line**.

```
puts "Hello!"     # => Hello!
print "Hi"        # => Hi
print " there!"   # => Hi there!
```

Escape Characters

Escape characters start with a backslash (`\`) and are used to include **special characters** inside strings.

Escape	Meaning	Example
`\n`	Newline	`"Line 1\nLine 2"`
`\t`	Tab space	`"Name:\tAlice"`
`\\`	Backslash (`\`)	`"C:\\Users\\Alice"`
`\"`	Double quote	`"He said, \"Hello!\""`
`\'`	Single quote	`'It\'s fine.'`

```
puts "She said, \"Welcome!\"\nLet’s start learning Ruby.\n"
```

Output

```
She said, "Welcome!"
Let’s start learning Ruby.
```

Input

To get input from the user, we use the `gets` method. It waits for the user to type something and press Enter.

```
puts "What's your name?"
name = gets
puts "Nice to meet you, #{name}!"
```

`gets` also captures the newline (`\n`) when the user presses Enter. To remove it, we use `.chomp`:

```
puts "What's your name?"
name = gets.chomp
puts "Nice to meet you, #{name}!"
```

Getting Numeric Input

All input from `gets` is treated as a **string** by default. If we want a number, we need to convert it using `.to_i` (for integers) or `.to_f` (for floats):

```
puts "Enter your age:"
age = gets.chomp.to_i
puts "In 5 years, you’ll be #{age + 5} years old!"
```

Tasks

Task 1

Write a program that reads the radius of a circle from the user and then displays its surface area.

Task 2

Develop a temperature converter that convert from celsius to fahrenheit