Objectives

- Object-Oriented Programming (OOP)
- Error Handling

Object Oriented Programming

Introduction

Object-Oriented Programming (OOP) is a programming paradigm that organizes software around "objects" that contain both data (attributes) and code (methods) that operate on that data.

- Objects: Represent real-world entities like a car, a person, or a bank account. Each object has unique characteristics (attributes) and behaviors (methods).
- Classes: Blueprints or templates for creating objects. A class defines the attributes and methods that objects of that class will have.

Core Principles of OOP:

- **Encapsulation**: Combines data and the functions that operate on that data within a single unit or class. This keeps data safe from outside interference and misuse.
- **Abstraction**: Allows programmers to hide all but the relevant data about an object to reduce complexity and increase efficiency.
- **Inheritance**: Facilitates the creation of new classes based on existing ones. It promotes code reusability by allowing shared behaviors to be defined once and inherited by child classes.
- **Polymorphism**: Enables different classes to be treated through the same interface, often by overriding methods or implementing the same methods in different ways.

Creating Class

In Ruby, we create a class using the `class` keyword followed by the class name. By convention, we use **CamelCase** for class names, meaning the first letter of each word is capitalized (e.g., `BankAccount`, `UserProfile`). Inside the class, we define methods that determine the object's behavior. If we want to create attributes (instance variables) that store an object's state, we prefix them with `@` (e.g., `@name`, `@balance`). These instance variables are private by default and can be accessed within any instance method of the class.

```
class Person
  @name = "Ali"
  @age = 20
  def greet
    puts "Hello, my name is #{@name}!"
  end
end
```

To create an object (instance) from a class in Ruby, we use the class name followed by the `.new` method.

```
preson = Person.new
person.greet()
```

Constructor

A constructor is a special method that runs automatically every time we create a new instance of a class. We use constructors primarily to initialize an object's attributes with starting values. In Ruby, we define a constructor by creating an `initialize` method within our class. This method is called automatically when we use the `new` method to create an object.

```
class Person
  def initialize(name, age)
    @name = name
    @age = age
  end
  def greet
    puts "Hello, my name is #{@name}!"
  end
end

person = Person.new("Alice", 30)
```

Getter and Setter

In Ruby, instance variables (attributes) are private by default. This means we cannot directly access them outside their class.If you try to access an attribute like this:

```
puts person.name
```

we will get a `NoMethodError` because the attribute isn't publicly accessible.

Controlled Access with Getters and Setters

To properly expose attributes, we use:

- **Getters** (read access)
- **Setters** (write access)

Ruby provides three convenient methods to define these:

1. `attr_reader` : Creates a getter (read-only access)
2. `attr_writer` : Creates a setter (write-only access)
3. `attr_accessor` : Creates both getter and setter

```
class Person
  attr_reader :age
  attr_writer :address
  attr_accessor :name

  def initialize(name, age)
    @name = name
    @age = age
  end
end

person = Person.new("Alice", 30)
puts person.name
person.name = "Bob"
puts person.age
person.age = 31
```

Instance Methods and Attributes

When designing classes, we sometimes need functionality that doesn't require creating an instance. Class variables, marked with double `@@` symbols, are shared across all instances of a class and can store values common to the entire class. We access these class variables using the `::` notation instead of dot notation. For example, a `Math` class might use `@@PI` as a class variable to store the mathematical constant.
Class methods are defined by prefixing the method name with `self.`, allowing them to be

called directly on the class without creating an instance. These methods are particularly useful for operations that don't depend on specific instance data.

```
 class MathUtils
  @@PI = 3.14159  # Class variable

  def self.circle_area(radius)
    @@PI * radius**2
  end
end

puts MathUtils::PI
puts MathUtils.circle_area(5)
```

Inheritance

To understand **inheritance**, let’s imagine we’re building a simple program to represent a **pet store**. In our store, we want to include different animals let’s say **cats** and **dogs**.
At first, we might think of creating two separate classes: `Cat` and `Dog`. But very quickly, we’ll notice that both animals share some common characteristics they both have names, ages, and they both can eat and sleep.
Instead of writing the same code twice, we can create a **parent class** called `Pet` that contains all the shared attributes and behaviors. Then, `Cat` and `Dog` can **inherit** from `Pet`, gaining all of its features, and we can add species-specific behavior to each one.

```
class Pet
  attr_accessor :name, :age
  def initialize(name, age)
    @name = name
    @age = age
  end
  def eat
    puts "#{name} is eating."
  end
  def sleep
    puts "#{name} is sleeping."
  end
end

class Dog < Pet
```

```
  def speak
    puts "#{name} says Woof!"
  end
end

class Cat < Pet
  def speak
    puts "#{name} says Meow!"
  end
end



dog = Dog.new("Buddy", 3)
cat = Cat.new("Whiskers", 2)
dog.eat
dog.speak
cat.sleep
cat.speak
```

We make a class **inherit** from a parent class using the `<` operator. This tells Ruby that the child class should have access to all the public and protected methods and attributes of the parent class.
Sometimes, we want the child class to **override** a method from the parent class but still keep part of the original behavior. In such cases, we can use the `super` keyword. This allows us to call the **parent class's version** of the method from inside the **child class** method.

```
class Pet
  attr_accessor :name, :age
  def initialize(name, age)
    @name = name
    @age = age
  end
  def speak
    puts "#{name} makes a sound."
  end
  def info
    puts "#{name} is #{age} years old."
  end
end
```

```
class Dog < Pet
  def speak
    super
    puts "#{name} says Woof!"
  end
end

class Cat < Pet
  def speak
    super
    puts "#{name} says Meow!"
  end
end

dog = Dog.new("Rex", 4)
cat = Cat.new("Luna", 2)
dog.speak
cat.speak
```

Modules & Mixins

In Ruby, we can **share functionality** between classes using **modules**. Modules are like containers for methods that we can “mix in” to our classes. We do this using the `include` or `extend` keywords.

- `include` Mixes in module methods as instance methods
- `extend` Mixes in module methods as class methods

```
module Greetings
  def hello
    puts "Hello!"
  end
end

class Person
  include Greetings
end
```

```
class Robot
  extend Greetings
end

person = Person.new
person.hello
Robot.hello
```

Encapsulation

To understand encapsulation, let’s continue working with our pet store example. When building a class like `Pet`, we want to make sure that internal details like how the pet's data is stored or updated are protected from the outside world
Encapsulation is all about controlling access to the data and behavior inside a class. In Ruby, we do this by using access control keywords: `public`, `private`, and `protected`.
By default, all methods in Ruby are public, which means they can be called from outside the class. But if we want to hide certain methods, we can mark them as `private`. This way, those methods can only be used within the class itself.

```
class Pet
  attr_accessor :name, :age

  def initialize(name, age)
    @name = name
    @age = age
  end

  def info
    puts "#{name} is #{age} years old."
    check_health
  end

  private

  def check_health
    puts "#{name}'s health is good!"
  end
end
```

```
pet = Pet.new("Coco", 4)
pet.info
```

`public` **methods** are accessible from anywhere both inside and outside the class. These are the methods we intend to use from other parts of our program. By default, all methods in a Ruby class are public unless we explicitly specify otherwise.
`private` **methods** are only accessible from within the same instance of the class. We can't call a private method with an explicit receiver (not even `self`). We typically use private methods for internal logic that we don't want to expose or allow other parts of the program to access directly.
`protected` **methods** sit somewhere between public and private. We can't call them from outside the class, but they can be accessed between instances of the same class or subclasses. Protected methods are handy when we want to let objects of the same type interact with each other like comparing two pets without fully exposing that behavior to everything else.

Polymorphism

Polymorphism means "many forms." In programming, it allows us to use the same method name to perform different behaviors depending on the object that's calling it. This makes our code more flexible and easier to extend.
Let's go back to our pet store example. Suppose we want each pet to "speak." We can define a `speak` method in both the `Dog` and `Cat` classes. Even though the method name is the same, each class can implement it differently. So when we call `speak` on a dog, it might bark and when we call it on a cat, it might meow.
Thanks to polymorphism, we can treat different objects in a uniform way. We can loop through a list of pets and call `speak` on each one, without worrying about whether it's a dog or a cat. Ruby will automatically run the correct version of the method based on the object's class.

```
class Pet
  attr_accessor :name
  def initialize(name)
    @name = name
  end
end

class Dog < Pet
  def speak
    puts "#{name} says Woof!"
  end
end

class Cat < Pet
```

```
  def speak
    puts "#{name} says Meow!"
  end
end

pets = [Dog.new("Rex"), Cat.new("Luna")]

pets.each do |pet|
  pet.speak
end
```

Operator Overloading

Operator overloading is the act of overriding basic operators like `+`, `-`, `*`, `==`, and others to work with our own classes. By default, Ruby doesn't know how to use these operators with custom objects. But by defining special methods in our class, we can tell Ruby what these operators should do.
Let’s say we’re working with a `FoodPortion` class in our pet store. We want to be able to **add** two portions together or **compare** if they are equal. Instead of creating separate methods like `add` or `equals`, we can overload the `+` and `==` operators to keep our code clean and natural.

```
class FoodPortion
  attr_reader :grams

  def initialize(grams)
    @grams = grams
  end

  # Overload + operator
  def +(other)
    FoodPortion.new(@grams + other.grams)
  end

  # Overload == operator
  def ==(other)
    @grams == other.grams
  end

  def to_s
    "#{grams}g"
```

```
  end
end

portion1 = FoodPortion.new(50)
portion2 = FoodPortion.new(30)
total = portion1 + portion2

puts "Total: #{total}"
puts portion1 == portion2
```

The `other` variable represents the second operand that we are applying the operator to.
We should always include it as a parameter for operators that require two operands.
Here's a small list of some operators that we can override:

Operator	Method to Define	Description
`+`	`def + (other)`	Addition
`-`	`def - (other)`	Subtraction
`*`	`def * (other)`	Multiplication
`/`	`def / (other)`	Division
`%`	`def % (other)`	Modulo (remainder)
`**`	`def ** (other)`	Exponentiation
`==`	`def == (other)`	Equality comparison
`!=`	`def != (other)`	Inequality (define `==` and Ruby handles this)
`<`	`def < (other)`	Less than
`<=`	`def <= (other)`	Less than or equal
`>`	`def > (other)`	Greater than
`>=`	`def >= (other)`	Greater than or equal

Error Handling

Introduction

When writing scripts and code, errors are inevitable. These errors can be **syntax errors**, such as misspelling a function or variable name, or **logical errors**, like mistakes in `if` conditions or loops that lead to unexpected behavior. Such errors can cause our program to fail, requiring fixes before the script can run properly.
Another category is **runtime errors**, which occur during script execution. These can be caused

by invalid user input or unpredictable conditions. While we can't always prevent runtime errors, JavaScript provides tools to handle them gracefully.

Types of Errors

- **Syntax Errors:** These occur when we make mistakes in writing our script, such as misspelling a function or variable name, or forgetting parentheses or curly braces.
- **Runtime Errors:** These errors occur while our program is running and cause it to stop. They are often caused by user input, such as entering invalid values.
- **Logic Errors:** These errors happen when the code does not work as intended due to incorrect formatting or logic in conditions. They don't crash the program but cause it to behave in unexpected ways.
- **Arithmetic Errors:** These are similar to logic errors but specifically involve mathematical operations, such as dividing by zero, or improper use of operator precedence (e.g., `+`, `-`, `/`, `*`).
- **Resource Errors:** These errors occur when we exceed the resources available on our machine, such as running out of memory or creating an infinite loop.

Ruby Errors

Error Type	Description	Example
`SyntaxError`	Invalid code structure	`def method( puts "hello"`
`NameError`	Accessing undefined variables	`puts notDefined`
`TypeError`	Wrong data type usage	`nil.to_s`
`RangeError`	Numeric value out of range	`Array.new(-1)`
`Custom Errors`	Developer-defined exceptions	`raise "Invalid input"`

Handling Runtime Errors

Some errors are caused by user input, which means that even if our code is clean, there could still be corner cases where user input causes the program to crash. To handle such cases, Ruby provides a way to catch errors and run alternative statements.
If we have a block of code that we know might cause errors, we put it inside the `begin` block. Then, to catch and handle the errors, we use the `rescue` keyword followed by the error type. If we write `rescue` without specifying an error type, it will capture all errors.

1. **`begin` Block:** The code that may cause an error is written inside the `begin` block.
2. **`rescue` Block:** After the `begin` block, the `rescue` block handles any errors that occur. We can specify the type of error to catch, or if no type is specified, it will catch all standard

errors.

3. `ensure` **Block (Optional):** The `ensure` block is optional and contains code that will always execute, regardless of whether an error occurred. It's like the `finally` block in other languages.

```
begin
  puts "Enter a number: "
  num = gets.chomp.to_i
  puts "You entered: #{num}"
rescue ArgumentError => e
  puts "Invalid input! Please enter a valid number."
  puts "Error details: #{e.message}"
rescue StandardError => e
  puts "An error occurred: #{e.message}"
ensure
  puts "Execution complete."
end
```

Raising Our Own Errors

We can raise our own exceptions using the `raise` keyword. This is useful when we want to enforce certain rules in your program.

```
def withdraw(balance, amount)
  if amount > balance
    raise "Insufficient funds"
  else
    puts "Withdrawing $#{amount}"
  end
end

withdraw(100, 50)
withdraw(100, 150)
```

Custom Exception Classes

We can create our own exception classes by inheriting from `StandardError`:

```
class OverdrawError < StandardError; end
```

```
def withdraw(balance, amount)
  raise OverdrawError, "Not enough money!" if amount > balance
  puts "Withdrawal successful"
end

begin
  withdraw(100, 200)
rescue OverdrawError => e
  puts "Custom error: #{e.message}"
end
```

Tasks:

Task 1:

Define a `BankAccount` class with:
Attribues:

- `accountNumber` A unique account identifier.
- `ownerName` The account holder's name.
- `balance` The current balance (default: `0`).
- `transactions` Stores transaction history.

Methods:

- **`initialize(accountNumber, ownerName, initialBalance = 0)`**
  - Initializes the account with given details.
  - Throws an error if `initialBalance` is negative.
- **`deposit(amount)`**
  - Adds `amount` to the balance.
  - Throws an error if `amount` is negative.
  - Logs the transaction in `#transactions`.
- **`withdraw(amount)`**
  - Deducts `amount` from the balance.
  - Throws an error if:
    - `amount` is negative.
    - Insufficient funds (`balance < amount`).
  - Logs the transaction in `#transactions`.
- **`getBalance()`**
  - Returns the current balance.

- `getStatement()`
  - Prints all transactions in a formatted way.
- `static transfer(sourceAccount, targetAccount, amount)`
  - A **static method** that transfers `amount` from `sourceAccount` to `targetAccount`.
  - Throws an error if the transfer fails.

Use `begin` and `rescue` to handle:

- Negative deposits/withdrawals.
- Insufficient funds.
- Invalid transfers.

Task 2 :

Create program to represent functionality of robot the program should have the following Parent class it represent basic rebot it have the following methods and attributes :
Attributes :

- **id :** attribute represent the id of the rebot
- **x :** represent the cordonite of robot according x axis
- **y :** represent the cordonite of robot according y axis
- **orientation :** the orientation of the robot and how it turned
- **step :** represent how much step the robot take when walking

Methodes :

- **constructor :** that inistalise the robot position x ,y , and it id, step is set by default as 1 no need to pass it as argument
- **turn_clockWise :** make the robot turn 90 degree clock wise direction
- **turn_antiClockWise :** make the robot turn -90 degree clock wise direction
- **walk :** makre rebot take step and change it x and y depend on it orientation
- **get_position :** print the position of the robot

Child class represent new generation of rebot, it inherite from the First generation of robot and have those additional feature
Attribues:

- **charge :** how much energie the rebot have
- **turbo_state :** it the robot is in turbo mode or not

Methodes:

- **constructor :** run the parent class contructor and after it assign value to charge, for turbo_state it set as False by `default`
- **turbo :** it turn the turbo mode on if charge > 0
- **walk :** if `turbo_state` is `False` it set `step` to 2 and remove 1 from `charge` else it return `step` to 1 and `turbo_state` to `False` then it run the parent class `walk`