20CYS312 - Principles of Programming Languages
Exploring Programming Paradigms

Assignment-01

Presented by NITIN G R
CB.EN.U4CYS21017
TIFAC-CORE in Cyber Security
Amrita Vishwa Vidyapeetham, Coimbatore Campus

Feb 2024

AMRITA
VISHWA VIDYAPEETHAM

Outline

Object-Oriented Programming (OOP)

- Object-Oriented Programming is a programming paradigm that relies on the concept of classes and objects.
- OOP is used to structure a software program into simple, reusable pieces of code blueprints (called classes), which are used to create individual instances of objects.
- The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Principles of Object-Oriented Programming

- Encapsulation: Encapsulation is the mechanism of hiding of data implementation by restricting access to public methods. Instance variables are kept private and accessor methods are made public to achieve this.
- Abstraction: Abstract means a concept or an Idea which is not associated with any particular instance. Using abstract class/Interface we express the intent of the class rather than the actual implementation.
- Inheritance: Inheritance allows classes to inherit features of other classes. Put another way, parent classes extend attributes and behaviors to child classes. Inheritance supports reusability.
- Polymorphism: It means one name many forms. Multiple classes can use the same method name using polymorphism, which also involves redefining methods for derived classes. Compile-time polymorphism and run-time polymorphism are the two different types of polymorphism.

Benefits of Object-Oriented Programming

- **Modularity:** Encapsulation enables objects to be self-contained, making troubleshooting and collaborative development easier.
- **Reusability:** Code can be reused through inheritance, meaning a team does not have to write the same code multiple times.
- **Security:** Using encapsulation and abstraction, complex code is hidden, software maintenance is easier, and internet protocols are protected.
- **Flexibility:** Polymorphism enables a single function to adapt to the class it is placed in. Different objects can also pass through the same interface.
- **Code Organization:** OOP promotes a structured approach, enhancing collaboration and code readability.

Ruby

- Ruby is an object-oriented programming language (OOP) that uses classes as blueprints for objects.
- Objects are the basic building-blocks of Ruby code (everything in Ruby is an object), and have two main properties: states and behaviours.
- Ruby classes are the blueprints that establish what attributes (also known as states) and behaviours (known in Ruby as methods) that an object should have.

Creating an Object from a Class

The Ruby `new` keyword is used to create an object belonging to a particular class. By doing so, memory is allotted for the object and a fresh instance of the class is initialized. For example, if we have a class called `Shape`, we can make an object of that class by using `Shape.new`. This creates a new object based on the class definition that is prepared to be used in our program.

```
class Shape
end

shape = Shape.new
```

How to Define a Class in Ruby?

In Ruby, the `class` keyword must be included before the class name to declare a class. A class definition allows for the specification of characteristics and methods of the class's objects. Methods define an object's behavior, whereas attributes show an object's current state. Consider the following example:

```
class Person
    def initialize(name)
        @name = name
    end

    def greet
        puts "Hello #{@name}"
    end
end

person = Person.new("Rohit")
person.greet # Output: Hello Rohit
```

Attribute Accessors

- `attr_reader`: Provides read-only access to an instance variable, allowing retrieval without external modifications.
- `attr_writer`: Restricts access to writing, enabling modification of the instance variable's value without direct retrieval; useful for external updates.
- `attr_accessor`: Combines `attr_reader` and `attr_writer`, allowing both reading and writing of an instance variable's value for total control.

```ruby
class Person

  attr_reader :name
  attr_writer :age
  attr_accessor :email

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

end
```

Inheritance

Creating a superclass that we'll call GreenSpace. The CityPark and Forest classes from this shared ancestor GreenSpace, both of those subclasses will inherit the behaviours

```ruby
class GreenSpace
  attr_reader :name, :num_trees

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

class CityPark < GreenSpace; end

class Forest < GreenSpace; end

high_park = CityPark.new("High Park", 5000)
durham_forest = Forest.new("Durham Forest", 125000)
dufferin_park = CityPark.new("Dufferin Park", 2000)

high_park.name # => "High Park"
high_park.num_trees # => 5000
durham_forest.name # => "Durham Forest"
durham_forest.num_trees # => 125000
dufferin_park.num_trees # => 2000
```

Module

Modules are Ruby's solution to multiple inheritance. It can be containers of methods.

```ruby
module Swimmable
  def swim
    "Can swim here!"
  end
end

class GreenSpace
  attr_reader :name, :num_trees

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

class CityPark < GreenSpace; end

class RecreationCentre < CityPark
  attr_reader :philanthropist
  include Swimmable

  def initialize(name, num_trees, philanthropist)
    super(name, num_trees)
    @philanthropist = philanthropist
  end
end
```

Polymorphism

Achieved through inheritance and the use of the 'duck typing'.

```ruby
class Animal
  def initialize(name)
    @name = name
  end

  def speak
    "#{@name} says "
  end
end

class Dog < Animal
  def speak
    super + 'Woof!'
  end
end

class Cat < Animal
  def speak
    super + 'Meow!'
  end
end

fido = Dog.new('Fido')
whiskers = Cat.new('Whiskers')
puts fido.speak  # Outputs: 'Fido says Woof!'
puts whiskers.speak  # Outputs: 'Whiskers says Meow!'
```

Abstraction

Abstraction in Ruby can be implemented using modules, which are like classes but cannot be instantiated. They're used to group related methods that can be included in multiple classes. Here's an example:

```ruby
module Movable
  def move
    puts "#{@name} is moving."
  end
end

class Animal
  include Movable

  def initialize(name)
    @name = name
  end
end

fido = Animal.new('Fido')
fido.move  # Outputs: 'Fido is moving.'
```

Aspect-Oriented

- Aspect-oriented programming (AOP) is a programming paradigm that aims to improve the modularity of software by allowing the separation of cross-cutting concerns.
- In other words, AOP helps to decompose complex problems into smaller, more manageable pieces by identifying and addressing common concerns that cut across the entire application.
- AOP is based on the concept of “aspects,” which are modular units that represent a particular concern or feature of the application.
- These aspects can be woven into the application code at specific points, known as “join points,” to add new behavior or modify existing behavior.

Principles of Aspected-Oriented Programming

- **Aspect:** An aspect is the part of the application which cross-cuts the core concerns of multiple objects. In Flow, aspects are implemented as regular classes which are tagged by the aspect annotation. The methods of an aspect class represent advices, the properties may be used for introductions.
- **Advice:** An advice is the action taken by an aspect at a particular join point. Advices are implemented as methods of the aspect class. These methods are executed before and / or after the join point is reached.
- **Introduction:** An introduction redeclares the target class to implement an additional interface. By declaring an introduction it is possible to introduce new interfaces and an implementation of the required methods without touching the code of the original class
- **Pointcut:** The pointcut defines a set of join points which need to be matched before running an advice. The pointcut is configured by a pointcut expression which defines when and where an advice should be executed. Flow uses methods in an aspect class as anchors for pointcut declarations.
- **Target:** A class or method being adviced by one or more aspects is referred to as a target class or method.

Benefits of Aspect-oriented programming

- Modularity: AOP helps to improve the modularity of software by allowing developers to separate cross-cutting concerns into modular units known as aspects. This can make it easier to understand, modify, and reuse code.
- Maintainability: By separating concerns, AOP can make it easier to maintain and modify code over time. This can be particularly useful when adding new features or behavior to an application without modifying the existing code.
- Reusability: Aspects can be reused across different projects, which can help to improve the efficiency and productivity of developers.
- Performance: AOP can be used to optimize the performance of an application by identifying and addressing bottlenecks or other inefficiencies.

JBoss AOP

- JBoss AOP is a 100% Pure Java Aspected Oriented Framework usable in any programming environment or tightly integrated with our application server.
- Aspects allow you to more easily modularize your code base when regular object oriented programming just does not fit the bill.
- It can provide a cleaner separation from application logic and system code. It provides a great way to expose integration points into your software.
- Combined with Java Annotations, it also is a great way to expand the Java language in a clean pluggable way rather than using annotations solely for code generation.
- JBoss AOP is not only a framework, but also a prepackaged set of aspects that are applied via annotations, pointcut expressions, or dynamically at runtime.
- Some of these include caching, asynchronous communication, transactions, security, remoting, and many more.
- An aspect is a common feature that is typically scattered across methods, classes, object hierarchies, or even entire object models. It is behavior that looks and smells like it should have structure, but you can not find a way to express this structure in code with traditional object-oriented techniques

Creating Aspects in JBoss AOP

The first step in creating a metrics aspect in JBoss AOP is to encapsulate the metrics feature in its own Java class. The following code extracts the try/finally block in our first code example's BankAccountDAO.withdraw() method into Metrics, an implementation of a JBoss AOP Interceptor class.

```
public class Metrics implements org.jboss.aop.advice.Interceptor
{
  public Object invoke(Invocation invocation) throws Throwable
  {
    long startTime = System.currentTimeMillis();
    try
    {
      return invocation.invokeNext();
    }
    finally
    {
      long endTime = System.currentTimeMillis() - startTime;
      java.lang.reflect.Method m = ((MethodInvocation)invocation).method;
      System.out.println("method " + m.toString() + " time: " + endTime + "
    }
  }
}
```

Applying Aspects in JBoss AOP

To apply an aspect, you define when to execute the aspect code. Those points in execution are called pointcuts. An analogy to a pointcut is a regular expression. Where a regular expression matches strings, a pointcut expression matches events or points within your application. For example, a valid pointcut definition would be, "for all calls to the JDBC method executeQuery(), call the aspect that verifies SQL syntax."
The following listing demonstrates defining a pointcut for the Metrics example in JBoss :

```
<interceptor name="SimpleInterceptor" class="com.mc.Metrics"/>
<bind pointcut="execution (public void com.mc.BankAccountDAO->withdraw(do
    <interceptor-ref name="SimpleInterceptor" />
</bind>
<bind pointcut="execution (* com.mc.billing.->(..))">
    <interceptor-ref name="com.mc.Metrics" />
</bind>
```

Paradigm 1: Object-Oriented Programming (OOP) in Ruby

Similarities with Aspect-Oriented Programming (AOP):

- Modularity: Both paradigms aim to improve code modularity. In OOP, modularity is achieved through classes and objects, while in AOP, it's achieved by separating cross-cutting concerns into aspects.
- Encapsulation: OOP promotes encapsulation by bundling data and methods that operate on the data within a class. Similarly, AOP encapsulates cross-cutting concerns in aspects.

Differences with Aspect-Oriented Programming (AOP):

- Inheritance vs. Cross-Cutting Concerns: OOP relies heavily on inheritance for code reuse and structure. In contrast, AOP focuses on addressing cross-cutting concerns like logging, security, and performance that may not fit neatly into a class hierarchy.
- Aspect vs. Class: In OOP, you create classes to represent objects and their behavior. In AOP, you create aspects to encapsulate cross-cutting concerns, which are then applied to different parts of the code.

Paradigm 2: Aspect-Oriented Programming (AOP) with JBoss AOP

Similarities with Object-Oriented Programming (OOP):

- Modularity: Both paradigms aim for modularity, but AOP achieves it by separating cross-cutting concerns, while OOP achieves it through class-based modularity.
- Encapsulation: AOP encapsulates cross-cutting concerns in aspects, similar to how OOP encapsulates data and behavior within classes.

Differences with Object-Oriented Programming (OOP):

- Cross-Cutting Concerns vs. Class Hierarchy: AOP focuses on cross-cutting concerns that affect multiple classes, whereas OOP often involves creating class hierarchies to model relationships and behaviors.
- Pointcuts and Join Points: AOP introduces concepts like pointcuts and join points to specify when and where aspects should be applied, which are not present in traditional OOP.

References

- www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP
- www.trio.dev/blog/object-oriented-programming
- emeritus.org/blog/coding-what-is-object-oriented-programming/
- www.pluralsight.com/resources/blog/cloud/what-is-the-ruby-programming-language
- www.scaler.com/topics/ruby/oops-in-ruby/
- medium.com/launch-school/the-basics-of-oop-ruby-26eaa97d2e98
- spiceworks.com/tech/devops/articles/what-is-aop/
- medium.com/hprog99/aspect-oriented-programming-b9a06ca256db