**Object Oriented Programming & Inheritance**

Programming languages have procedural programming, functional programming and object-oriented programming paradigms. Ruby is an object-oriented language with some functional and procedural features.

*Object-oriented programming (OOP)* is a programming paradigm that uses objects and their interactions to design applications and computer programs.

The basic programming concepts in OOP are:

* Abstraction
* Polymorphism
* Encapsulation
* Inheritance

The *abstraction* is simplifying complex reality by modeling classes appropriate to the problem. The*polymorphism* is the process of using an operator or function in different ways for different data input. The *encapsulation* hides the implementation details of a class from other objects. The*inheritance* is a way to form new classes using classes that have already been defined.

**Objects**

Objects are basic building blocks of a Ruby OOP program. An object is a combination of data and methods. In a OOP program, we create objects. These objects communicate together through methods. Each object can receive messages, send messages and process data.

There are two steps in creating an object. First, we define a class. A *class* is a template for an object. It is a blueprint that describes the state and behavior that the objects of the class all share. A class can be used to create many objects. Objects created at runtime from a class are called *instances* of that particular class.

#!/usr/bin/ruby

class Being

end

b = Being.new

puts b

In our first example, we create a simple object.

class Being

end

This is a simple class definition. The body of the template is empty. It does not have any data or methods.

b = Being.new

We create a new instance of the Being class. For this we have the new method. The b variable stores the newly created object.

puts b

We print the object to the console to get some basic description of the object. When we print an object, we in fact call its to\_s method. But we have not defined any method yet. It is because every object created inherits from the base Object. It has some elementary functionality, which is shared among all objects created. One of this is the to\_s method.

$ ./simple.rb

#<Being:0x9f3c290>

We get the object class name.

**The constructor**

A constructor is a special kind of a method. It is automatically called when an object is created. Constructors do not return values. The purpose of the constructor is to initiate the state of an object. The constructor in Ruby is called initialize. Constructors do not return any values.

Constructors cannot be inherited. The constructor of a parent object is called with a super method. They are called in the order of inheritance.

#!/usr/bin/ruby

class Being

def initialize

puts "Being is created"

end

end

Being.new

We have a Being class.

class Being

def initialize

puts "Being is created"

end

end

The Being class has a constructor method called initialize. It prints a message to the console. The definition of a Ruby method is placed between the def and end keywords.

Being.new

An instance of the Being class is created. At the moment of the object creation, the constructor method is called.

$ ./constructor.rb

Being is created

This is the output of the program.

An object's attributes are the data items that are bundled inside that object. These items are also called *instance variables* or *member fields*. An instance variable is a variable defined in a class, for which each object in the class has a separate copy.

In the next example, we initiate data members of the class. Initiation of variables is a typical job for constructors.

#!/usr/bin/ruby

class Person

def initialize name

@name = name

end

def get\_name

@name

end

end

p1 = Person.new "Jane"

p2 = Person.new "Beky"

puts p1.get\_name

puts p2.get\_name

In the above Ruby code, we have a Person class with one member field.

class Person

def initialize name

@name = name

end

In the constructor of the Person class, we set a member field to a value name. The name parameter is passed to the constructor at creation. A constructor is a method called initialize that is being called at the creation of an instance object. The @name is an instance variable. Instance variables start with @ character in Ruby.

def get\_name

@name

end

The get\_name method returns the member field. In Ruby member fields are accessible only via methods.

p1 = Person.new "Jane"

p2 = Person.new "Beky"

We create two objects of a Person class. A string parameter is passed to each object constructor. The names are stored inside instance variables that are unique to each object.

puts p1.get\_name

puts p2.get\_name

We print the member fields by calling the get\_name on each of the objects.

$ ./person.rb

Jane

Beky

We see the output of the program. Each instance of the Person class has its own name member field.

We can create an object without calling the constructor. Ruby has a special allocate method for this. The allocate method allocates space for a new object of a class and does not call initialize on the new instance.

#!/usr/bin/ruby

class Being

def initialize

puts "Being created"

end

end

b1 = Being.new

b2 = Being.allocate

puts b2

In the example, we create two objects. The first object using the new method, the second object using the allocate method.

b1 = Being.new

Here we create an instance of the object with the new keyword. The constructor method initialize is called and the message is printed to the console.

b2 = Being.allocate

puts b2

In case of the allocate method, the constructor is not called. We call the to\_s method with the putskeyword to show that the object was created.

$ ./allocate.rb

Being created

#<Being:0x8ea0044>

Here we see the output of the program.

**Constructor overloading**

Constructor overloading is the ability to have multiple types of constructors in a class. This way we can create an object with different number or different types of parameters.

Ruby *has no constructor overloading* that we know from some programming languages. This behaviour can be simulated to some extent with default parameter values in Ruby.

#!/usr/bin/ruby

class Person

def initialize name="unknown", age=0

@name = name

@age = age

end

def to\_s

"Name: #{@name}, Age: #{@age}"

end

end

p1 = Person.new

p2 = Person.new "unknown", 17

p3 = Person.new "Becky", 19

p4 = Person.new "Robert"

p p1, p2, p3, p4

This example shows how we could simulate constructor overloading on a Person class that has two member fields. When the name parameter is not specified, the string "unknown" is used instead. For unspecified age we have 0.

def initialize name="unknown", age=0

@name = name

@age = age

end

The constructor takes two parameters. They have a default value. The default value is used if we do not specify our own values at the object creation. Note that the order of parameters must be kept. First comes the name, then the age.

p1 = Person.new

p2 = Person.new "unknown", 17

p3 = Person.new "Becky", 19

p4 = Person.new "Robert"

p p1, p2, p3, p4

We create four objects. The constructors take different number of parameters.

$ ./consover.rb

Name: unknown, Age: 0

Name: unknown, Age: 17

Name: Becky, Age: 19

Name: Robert, Age: 0

This is the output of the example.

**Methods**

Methods are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects. Methods are essential in the *encapsulation* concept of the OOP paradigm. For example, we might have a connect method in our AccessDatabase class. We need not to be informed how exactly the method connects to the database. We only have to know that it is used to connect to a database. This is essential in dividing responsibilities in programming, especially in large applications.

In Ruby, data is accessible only via methods.

#!/usr/bin/ruby

class Person

def initialize name

@name = name

end

def get\_name

@name

end

end

per = Person.new "Jane"

puts per.get\_name

puts per.send :get\_name

The example shows two basic ways to call a method.

puts per.get\_name

The common way is to use a dot operator on an object followed by a method name.

puts per.send :get\_name

The alternative is to use a built-in send method. It takes a symbol of the method to be called as a parameter.

Methods typically perform some action on an object's data.

#!/usr/bin/ruby

class Circle

@@PI = 3.141592

def initialize

@radius = 0

end

def set\_radius radius

@radius = radius

end

def area

@radius \* @radius \* @@PI

end

end

c = Circle.new

c.set\_radius 5

puts c.area

In the code example, we have a Circle class. We define two methods.

@@PI = 3.141592

We have defined a @@PI variable in our Circle class. It is a class variable. Class variables start with @@sigils in Ruby. Class variables belong to a class, not to an object. Each object has access to its class variables. We use the @@PI to compute the area of the circle.

def initialize

@radius = 0

end

We have one member field. It is the radius of the circle. If we want to modify this variable from the outside, we must use the publicly available set\_radius method. The data is protected.

def set\_radius radius

@radius = radius

end

This is the set\_radius method. It gives the @radius instance variable a new value.

def area

@radius \* @radius \* @@PI

end

The area method returns the area of a circle. This is a typical task for a method. It works with data and produces some value for us.

c = Circle.new

c.set\_radius 5

puts c.area

We create an instance of the Circle class, and set its radius by calling the set\_radius method on the object of the circle. We use the dot operator to call the method.

$ ./circle.rb

78.5398

Running the example we get this output.

**Access modifiers**

Access modifiers set the visibility of methods and member fields. Ruby has three access modifiers: public, protected and private. In Ruby, all data members are private. Access modifiers can be used only on methods. Ruby methods are public, unless we say otherwise.

The public methods can be accessed from inside the definition of the class as well as from the outside of the class. The difference between the protected and the private methods is subtle. Neither can be accessed outside the definition of the class. They can be accessed only within the class itself and by inherited or parent classes.

Note that unlike in other object-oriented programming languages, *inheritance does not play role* in Ruby access modifiers. Only two things are important. First, if we call the method inside or outside the class definition. Second, if we use or do not use the self keyword which points to the current receiver.

Access modifiers protect data against accidental modifications. They make the programs more robust. The implementation of some methods is subject to change. These methods are good candidates for being private. The interface that is made public to the users should only change when really necessary. Over the years users are accustomed to using specific methods and breaking backward compatibility is generally frowned upon.

#!/usr/bin/ruby

class Some

def method1

puts "public method1 called"

end

public

def method2

puts "public method2 called"

end

def method3

puts "public method3 called"

method1

self.method1

end

end

s = Some.new

s.method1

s.method2

s.method3

The example explains the usage of public Ruby methods.

def method1

puts "public method1 called"

end

The method1 is public, even if we did not specify the public access modifier. It is because methods are public by default if not specified otherwise.

public

def method2

puts "public method2 called"

end

...

Methods following the public keyword are public.

def method3

puts "public method3 called"

method1

self.method1

end

From inside the public method3, we call other public method, with and without the self keyword.

s = Some.new

s.method1

s.method2

s.method3

Public methods are the only methods that can be called outside the definition of a class as shown here.

$ ./public\_methods.rb

public method1 called

public method2 called

public method3 called

public method1 called

public method1 called

Running the example we have this output.

The next example looks at private methods.

#!/usr/bin/ruby

class Some

def initialize

method1

# self.method1

end

private

def method1

puts "private method1 called"

end

end

s = Some.new

# s.method1

Private methods are tightest methods in Ruby. They can be called only inside a class definition and without the self keyword.

def initialize

method1

# self.method1

end

In the constructor of the method, we call the private method1. Calling the method with the self is commented. Private methods cannot be specified with a receiver.

private

def method1

puts "private method1 called"

end

Methods following the private keyword are private in Ruby.

s = Some.new

# s.method1

We create an instance of the Some class. Calling the method outside the class definition is prohibited. If we uncomment the line, the Ruby interpreter gives an error.

$ ./private\_methods.rb

private method called

Output of the example code.

Finally, we will work with protected methods. The distinction between protected and private methods in Ruby is subtle. Protected methods are like private. There is only one small difference. They can be called with the self keyword specified.

#!/usr/bin/ruby

class Some

def initialize

method1

self.method1

end

protected

def method1

puts "protected method1 called"

end

end

s = Some.new

# s.method1

The above example shows protected method in usage.

def initialize

method1

self.method1

end

Protected methods can be called with and without the self keyword.

protected

def method1

puts "protected method1 called"

end

Protected methods are preceded by the protected keyword.

s = Some.new

# s.method1

Protected methods cannot be called outside the class definition. Uncommenting the line would lead to an error.

**Inheritance**

Inheritance is a way to form new classes using classes that have already been defined. The newly formed classes are called *derived* classes, the classes that we derive from are called *base* classes. Important benefits of inheritance are code reuse and reduction of complexity of a program. The derived classes (descendants) override or extend the functionality of base classes (ancestors).

#!/usr/bin/ruby

class Being

def initialize

puts "Being class created"

end

end

class Human < Being

def initialize

super

puts "Human class created"

end

end

Being.new

Human.new

In this program, we have two classes: a base Being class and a derived Human class. The derived class inherits from the base class.

class Human < Being

In Ruby, we use the < operator to create inheritance relations. The Human class inherits from the Beingclass.

def initialize

super

puts "Human class created"

end

The super method calls the constructor of the parent class.

Being.new

Human.new

We instantiate the Being and the Human class.

$ ./inheritance.rb

Being class created

Being class created

Human class created

First the Being class is created. The derived Human class also calls the constructor of its parent.

An Object may be involved in complicated relationships. A single object can have multiple ancestors. Ruby has a method ancestors which gives a list of ancestors for a specific class.

Each Ruby object is automatically a descendant of Object and BasicObject classes and of the Kernelmodule. They are built-in the core of the Ruby language.

#!/usr/bin/ruby

class Being

end

class Living < Being

end

class Mammal < Living

end

class Human < Mammal

end

p Human.ancestors

We have four classes in this example: a Human is a Mammal a Living and a Being.

p Human.ancestors

We print the ancestors of a Human class.

$ ./ancestors.rb

[Human, Mammal, Living, Being, Object, Kernel, BasicObject]

A Human class has three custom and three built-in ancestors.

A more complex example follows.

#!/usr/bin/ruby

class Being

@@count = 0

def initialize

@@count += 1

puts "Being class created"

end

def show\_count

"There are #{@@count} beings"

end

end

class Human < Being

def initialize

super

puts "Human is created"

end

end

class Animal < Being

def initialize

super

puts "Animal is created"

end

end

class Dog < Animal

def initialize

super

puts "Dog is created"

end

end

Human.new

d = Dog.new

puts d.show\_count

We have four classes. The inheritance hierarchy is more complicated. The Human and the Animalclasses inherit from the Being class. And the Dog class inherits directly from the Animal class and further from the Being class. We also use a class variable to count the number of beings created.

@@count = 0

We define a class variable. A class variable begins with @@ sigils and it belongs to the class, not to the instance of the class. We use it to count the number of beings created.

def initialize

@@count += 1

puts "Being class created"

end

Each time the Being class is instantiated, we increase the @@count variable by one. This way we keep track of the number of instances created.

class Animal < Being

...

class Dog < Animal

...

The Animal inherits from the Being and the Dog inherits from the Animal. Further, the Dog inherits from the Being as well.

Human.new

d = Dog.new

puts d.show\_count

We create instances from the Human and from the Dog classes. We call the show\_count method on theDog object. The Dog class has no such method; the grandparent's (Being's) method is called then.

$ ./inheritance2.rb

Being class created

Human is created

Being class created

Animal is created

Dog is created

There are 2 beings

The Human object calls two constructors. The Dog object calls three constructors. There are two beings instantiated.

Inheritance does not play role in the visibility of methods and data members. This is a notable difference from many common object-oriented programming languages.

In C# or Java, public and protected data members and methods are inherited; private data members and methods are not. In contrast to this, private data members and methods are inherited in Ruby as well. The visibility of data members and methods is not affected by inheritance in Ruby.

#!/usr/bin/ruby

class Base

def initialize

@name = "Base"

end

private

def private\_method

puts "private method called"

end

protected

def protected\_method

puts "protected\_method called"

end

public

def get\_name

return @name

end

end

class Derived < Base

def public\_method

private\_method

protected\_method

end

end

d = Derived.new

d.public\_method

puts d.get\_name

In the example we have two classes. The Derived class inherits from the Base class. It inherits all three methods and one data field.

def public\_method

private\_method

protected\_method

end

In the public\_method of the Derived class we call one private and one protected method. They were defined in the parent class.

d = Derived.new

d.public\_method

puts d.get\_name

We create an instance of the Derived class. We call public\_method, as well as also get\_name, which returns the private @name variable. Remember that all instance variables are private in Ruby. The get\_name method returns the variable, regardless of @name's being private and defined in the parent class.

$ ./inheritance3.rb

private method called

protected\_method called

Base

The output of the example confirms that in Ruby language, public, protected, private methods and private member fields are inherited by child objects from their parents.

**The super method**

The super method calls a method of the same name in the parent's class. If the method has no arguments it automatically passes all its arguments. If we write super() no arguments are passed to parent's method.

#!/usr/bin/ruby

class Base

def show x=0, y=0

p "Base class, x: #{x}, y: #{y}"

end

end

class Derived < Base

def show x, y

super

super x

super x, y

super()

end

end

d = Derived.new

d.show 3, 3

In the example, we have two classes in a hierarchy. They both have a show method. The show method in the Derived class calls the show method in the Base class using the super method.

def show x, y

super

super x

super x, y

super()

end

The super method without any arguments calls the parent's show method with the arguments that were passed to the show method of the Derived class: here, x=3 and y=3. The super() method passes no arguments to the parent's show method.

$ ./super.rb

"Base class, x: 3, y: 3"

"Base class, x: 3, y: 0"

"Base class, x: 3, y: 3"

"Base class, x: 0, y: 0"

This is the output of the example.