第四章　物件導向基礎

OOP with Ruby

本章內容

- 物件導向觀念
- 什麼是物件？
- 什麼是類別？
- 物件導向設計的三大特性： 資料封裝

物件導向觀念

- In object-oriented programming (OOP), the fundamental unit is the object
  - An object is an entity that serves as a container for data and also controls access to the data
  - Associated with an object is a set of attributes (or called instance variables, data member), which are essentially no more than variables belonging to the object
  - Also associated with an object is a set of functions that provide an interface to the functionality of the object. These functions are called methods

什麼是物件？什麼是類別？

- An object is considered to be an instance or manifestation of an object class (usually simply called a class)
  - The class may be thought of as the blueprint or pattern
  - The object itself is the thing created from that blueprint or pattern
  - A class is often thought of as an abstract type or a more complex type than primitive types such as integer, character, etc.
- A class is a special kind of programmer-defined type

什麼是物件？什麼是類別？(2)

- If a piece of data is more "global" in scope than a single object, and it is inappropriate to put a copy of the attribute into each instance of the class
  - This is called a class attribute or class variable
  - Ordinary attributes are called object attributes, instance variables, or data members
- It is worth mentioning that there is a sense in which all methods are class methods
  - We should not suppose that when a hundred objects are created, we actually copy the code for the methods a hundred times

Ruby 的物件導向特性

- Ruby has all the elements more generally associated with OOP languages
  - Objects with encapsulation and data hiding
  - Methods with polymorphism and overriding (ref. to Chap. 5)
  - Classes with hierarchy and inheritance (ref. to Chap. 5)
- In Ruby, all numbers, strings, arrays, regular expressions, and many other entities are actually objects
- Work is done by executing the methods belonging to the object

Ruby 的物件

```
3.succ                  # 4
"abc".upcase            # "ABC"
[2,1,5,3,4].sort        # [1,2,3,4,5]
someObject.someMethod   # some result
```

- In Ruby, every object is an instance of some class

```
"abc".type    # String
"abc".class   # String
```

- Every object in Ruby has an identity

```
"abc".id      #  53744407
```

Ruby 的物件 (2)

- Variables are used to hold references to objects

  yourString = "this is also a string object"

  aNumber = 5

物件導向設計的三大特性：資料封裝 (Encapsulation)

- It is essential that any OOP language provide encapsulation
  - The attributes and methods of an object are associated specifically with that object or bundled with it
  - The scope of those attributes and methods is by default the object itself
  - Similar to data hiding concept, but with methods also

建立物件

- To define a new class, the following construct is used

```
class ClassName
  # ...
end
```

  - The name of the class is itself a global constant and therefore must begin with an uppercase letter

- The class definition can contain class constants, class variables, class methods, instance variables, and instance methods

建立物件範例

- Ruby use “class” keyword to declare a class
  - A special method called “initialize” is the constructor of a Ruby class

```
class Song
  def initialize(name, artist, duration)
    @name     = name
    @artist   = artist
    @duration = duration
  end
end
```

  - The above class has three instance variable called “name”, “artist”, and “duration”

物件的建構

- You can use “new” method to create an object of a specific class

  song = Song.new

  – The initialize method is the constructor of class Song, which is a special kind of method designed to initialize the instance variable for an object

  – When Song.new is called, Ruby allocates memory to hold an uninitialized object and then calls that object’s initialize method, passing in any parameters that were passed to new.

  – Constructor gives the change to set up object’s state

```
song = Song.new("Bicylops", "Fleck", 260)
song.to_s   →   "#<Song:0x1c7ec4>"
```

定義方法

- A method is defined using the keyword def
- Method names should begin with lowercase letters

```
def my_new_method(arg1, arg2, arg3)     # 3 arguments
  # Code for the method would go here
end
def my_other_new_method                  # No arguments
  # Code for the method would go here
end
```

- Such methods are called instance methods

定義方法 (2)

- Ruby lets you specify default values for a method's argumens—values that will be use if the caller doesn't pass them explicitly
  - You do this usin the assignment operator

```
def cool_dude(arg1="Miles", arg2="Coltrane", arg3="Roach")
  "#{arg1}, #{arg2}, #{arg3}."
end

cool_dude                                 →   "Miles, Coltrane, Roach."
cool_dude("Bart")                         →   "Bart, Coltrane, Roach."
cool_dude("Bart", "Elwood")               →   "Bart, Elwood, Roach."
cool_dude("Bart", "Elwood", "Linus")      →   "Bart, Elwood, Linus."
```

定義方法 (3)

- If you want to pass in a variable number of arguments or want to capture multiple arguments into a single parameter

```
def varargs(arg1, *rest)
  "Got #{arg1} and #{rest.join(', ')}"
end

varargs("one")                    →   "Got one and "
varargs("one", "two")             →   "Got one and two"
varargs "one", "two", "three"     →   "Got one and two, three"
```

定義方法 (4)

- Methods that act as queries are often named with a trailing ?
  - Such as instance_of?
- Methods that are “dangerous,” or modify the receiver, may be named with a trailing !
  - String provides both a chop and a chop!. The first one returns a modified string; the second modifies the receiver in place

呼叫方法

- Method with parameter(s) can be written as a name with a parenthesis contains a list of parameter(s)
  - If no ambiguity exists, you can omit the parentheses around the argument list when calling a method

```
a = obj.hash     # Same as
a = obj.hash()   # this.
obj.some_method "Arg1", arg2, arg3   # Same thing as
obj.some_method("Arg1", arg2, arg3)  # with parentheses.
```

方法的傳回值

- Every called method returns a value
  - The value of a method is the value of the last statement executed during the method's execution
  - Ruby has a return statement, which exits from the currently executing method
  - "return" can be omitted

方法的傳回值 (2)

```
def meth_one
  "one"
end
meth_one   →   "one"

def meth_two(arg)
  case
  when arg > 0
    "positive"
  when arg < 0
    "negative"
  else
    "zero"
  end
end

meth_two(23)   →   "positive"
meth_two(0)    →   "zero"
```

讀取物件屬性

- Here we've defined three accessor methods to return the values of the three instance variables

```
class Song
  def name
    @name
  end
  def artist
    @artist
  end
  def duration
    @duration
  end
end
song = Song.new("Bicylops", "Fleck", 260)
song.artist     →   "Fleck"
song.name       →   "Bicylops"
song.duration   →   260
```

讀取物件屬性 (2)

- Ruby provides a convenient shortcut: attr_reader creates the accessor methods for you

```
class Song
  attr_reader :name, :artist, :duration
end
```

變更物件屬性

- In Ruby you do the attribute setting job by creating a method whose name ends with an equals sign

```
class Song
  def duration=(new_duration)
    @duration = new_duration
  end
end
song = Song.new("Bicylops", "Fleck", 260)
song.duration   →   260
song.duration = 257   # set attribute with updated value
song.duration   →   257
```

變更物件屬性 (2)

- Ruby provides a convenient shortcut: attr_reader creates the mutator (attribute-setting) methods for you

```
class Song
  attr_writer :duration
end
song = Song.new("Bicylops", "Fleck", 260)
song.duration = 257
```

同時定義屬性讀取與變更方法

- You can use attr_accessor instead of both attr_reader and attr_writer

虛擬屬性的建立

- Use attribute accessor or mutator-like method to represent a virtually not-existed instance variable

```
class Song
  def duration_in_minutes
    @duration/60.0   # force floating point
  end
  def duration_in_minutes=(new_duration)
    @duration = (new_duration*60).to_i
  end
end
song = Song.new("Bicylops", "Fleck", 260)
song.duration_in_minutes   →   4.33333333333333
song.duration_in_minutes = 4.2
song.duration              →   252
```

類別變數與類別方法

- So far we discussed variables that are associated with a particular instance of the class, and methods that work on those variables
  - We call them instance variables and instance methods
- Sometimes classes themselves need to have their own states

類別變數

- A class variable is shared among all objects of a class, and it is also accessible to the class methods
  - Only one copy o a particular class variable exists for a given class
  - Class variable names start with two “at” signs, such @@count
  - Unlike global and instance variables, class variables must be initialized before they are used

類別變數範例

```
class Song
  @@plays = 0
  def initialize(name, artist, duration)
    @name     = name
    @artist   = artist
    @duration = duration
    @plays    = 0
  end
  def play
    @plays  += 1   # same as @plays = @plays + 1
    @@plays += 1
    "This  song: #@plays plays. Total #@@plays plays."
  end
end
s1 = Song.new("Song1", "Artist1", 234)  # test songs..
s2 = Song.new("Song2", "Artist2", 345)
s1.play   →   "This  song: 1 plays. Total 1 plays."
s2.play   →   "This  song: 1 plays. Total 2 plays."
s1.play   →   "This  song: 2 plays. Total 3 plays."
s1.play   →   "This  song: 3 plays. Total 4 plays."
```

類別變數範例 (2)

- Class variables are private to a class and its instances
  - If you want to make them accessible to the outside world, you'll need to write an accessor method

類別方法

- Sometimes a class needs to provide methods that work without being tied to any particular object
  - The constructor is such a method
- Class methods are distinguished from instance methods by their definition
  - Class methods are defined by placing the class name and a period in front of the method name

```
class Example
  def instance_method           # instance method
  end
  def Example.class_method      # class method
  end
end
```

類別與物件的存取控制

- When designing a class interface, it's important to consider just how much access to your class you'll be exposing to the outside world
  - Allow to much access into your class, and you risk increasing the coupling in your application
  - A good rule of thumb is never to expose methods that could leave an object in an invalid state

類別與物件的存取控制 (2)

- Ruby gives you three levels of protection
  - Public methods can be called by anyone—no access control is enforced. Methods are public by default (except for initialize, which is always private)
  - Protected methods can be invoked only by objects of the defining class and its subclasses. Access is kept within the family.
  - Private methods cannot be called with an explicit receiver—the receiver is always *self*. This means that private methods can be called only in the context of the current object; you can't invoke another object's private methods.

如何定義存取方式

```
class MyClass
      def method1     # default is 'public'
        #...
      end
  protected           # subsequent methods will be 'protected'
      def method2     # will be 'protected'
        #...
      end
  private             # subsequent methods will be 'private'
      def method3     # will be 'private'
        #...
      end
  public              # subsequent methods will be 'public'
      def method4     # and this will be 'public'
        #...
      end
end
```

如何定義存取方式 (2)

- Alternatively, you can set access levels of named methods by listing them as arguments to the access control functions

```
class MyClass
  def method1
  end
  # ... and so on
  public    :method1, :method4
  protected :method2
  private   :method3
end
```

本章回顧