第五章　物件導向進階

OOP with Ruby

本章內容

- 繼承概念 (Inheritance)
- 多形概念 (Polymorphism)
- Ruby 的動態語言特性： 模組概念
- 行為決定類型的分類法 : duck-typing

繼承概念 (Inheritance)

- Inheritance is one of the main techniques of object-oriented programming (OOP)
  - Inheritance is a relation between two classes
- Using this technique, a very general form of a class is first defined and compiled, and then more specialized versions of the class are defined by adding instance variables and methods
  - The specialized classes are said to inherit the methods and instance variables of the general class

繼承概念 (2)

- Inheritance is the process by which a new class is created from another class
  - The new class is called a *derived* class or *subclass*
  - The original class is called the *base* class or *superclass*
- A derived class automatically has all the instance variables and methods that the base class has, and it can have additional methods and/or instance variables as well
- Inheritance is especially advantageous because it allows code to be reused, without having to copy it into the definitions of the derived classes

繼承概念 (3)

- In Ruby, a class can only inherit from a single other class
  - One class has only one base class
  - This is called *single inheritance*
- In Ruby, you express the inheritance relationship by using the symbol "<"

繼承範例

- All cats are mammals, all mammals are animals

```
class Mammal
  def breathe
    puts "inhale and exhale"
  end
end
class Cat < Mammal
  def speak
    puts "Meow"
  end
end
rani = Cat.new
rani.breathe
rani.speak
```

繼承範例 (2)

- Though we didn't specify how a Cat should breathe, every cat will inherit that behavior from the Mammal class since Cat was defined as a subclass of Mammal
  - From a programmer's standpoint, cats get the ability to breathe for free
  - After we add a speak method, our cats can both breathe and speak
- There will be situations where certain properties of the super-class should not be inherited by a particular subclass
  - For example, though birds generally know how to fly, penguins are a flightless subclass of birds

繼承範例 (3)

```
class Bird
  def preen
    puts "I am cleaning my feathers."
  end
  def fly
    puts "I am flying."
  end
end
class Penguin < Bird
  def fly
    puts "Sorry. I'd rather swim."
  end
end
p = Penguin.new
p.preen
p.fly
```

覆寫觀念 (override)

- Although a derived class inherits methods from the base class, it can change or override an inherited method if necessary
  - In the above example, we *override* fly in the Penguin class
- Rather than exhaustively define every characteristic of every new class, we need only to append or to redefine the differences between each subclass and its super-class
  - This use of inheritance is sometimes called differential programming

繼承的成員

- Definitions for the inherited variables and methods do not appear in the derived class
  - The code is reused without having to explicitly copy it, unless the creator of the derived class redefines one or more of the base class methods

Ruby 所沒有的覆載 (overload) 觀念

- Two or more methods with the same method name in a class is called "method overloading"
- Ruby don't have such overloading concept since the method arguments has variable length.

Ruby 所沒有的覆載觀念 (2)

```
def method_a(a)
    puts "method_a(a) "+a.to_s
end

def method_a(a,b)
    puts "method_a(a,b) "+a.to_s+" "+b.to_s
end

method_a("a")
# error found above since method_a has been redefined
method_a("a","b")
```

“super” 關鍵字

- A derived class uses a constructor from the base class to initialize all the data inherited from the base class
  - In order to invoke a constructor from the base class, it uses a special syntax

```
def initialize(p1, p2, p3)
  super(p1, p2)
  instanceVar = p3
end
```

  - In the above example, super(p1, p2) is a call to the base class constructor

多形概念 (Polymorphism)

- There are three main programming mechanisms that constitute object-oriented programming (OOP)
  - Encapsulation
  - Inheritance
  - Polymorphism
- Polymorphism is the ability to associate many meanings to one method name across class inheritance chain
  - It does this through a special mechanism known as late binding or dynamic binding

多形概念 (2)

- Inheritance allows a base class to be defined, and other classes derived from it
  - Code for the base class can then be used for its own objects, as well as objects of any derived classes
- Polymorphism allows changes to be made to method definitions in the derived classes, and have those changes apply to the software written for the base class

晚期繫結 (Late Binding)

- The process of associating a method definition with a method invocation is called binding
- If the method definition is associated with its invocation when the method is invoked (at run time), that is called late binding or dynamic binding
  - Ruby is a scripting language, the method invocation is naturally late binding

晚期繫結 (2)

- Because of late binding, a method can be written in a base class to perform a task, even if portions of that task aren't yet defined
- Polymorphism in Ruby is just a matter of matching method names

多形範例

```
class Sale
  attr_accessor :name
  attr_accessor :price
  def initialize(name, price)
    @name = name
    @price = price
  end
  def lessThan(otherSale)
    return bill() < otherSale.bill()
  end
  def bill
    @price
  end
end
```

多形範例 (2)

```
class DiscountSale < Sale
  attr_accessor :discount
  def initialize(name, price, discount)
    super(name, price)
    @discount = discount
  end
  def bill
    (1-discount/100)/price
  end
end
```

多形範例 (3)

```
simple = Sale.new("carpet", 10)

discount = DiscountSale.new("carpet", 11, 10)

if (discount.lessThan(simple))

  puts "$" + discount.bill().to_s + "< $" + simple.bill() + "because
    late-binding works."
```

- As the output indicates, when the lessThan method in the Sale class is executed, it knows which bill()method to invoke
  - Note that when the Sale class was created, the DiscountSale class and its bill() method did not yet exist

Ruby 的動態語言特性：模組概念

- Modules are a way of grouping together methods, classes, and constants
- Two benefits
  - Modules provide a *namespace* and prevent name clashes
  - Modules implement the *mixin* facility

命名空間 (Namespace)

- Reusable codes are often organized into files for later inclusion into program files
  - Use keyword “require”
  - What if we found two same name method in our “require”?
    - Bad news!
- Module define a namespace
  - A sandbox in which your methods and constants can play without having to worry about being stepped on by other methods and constants

模組範例

- Create trig.rb & moral.rb first

```
module Trig
  PI = 3.141592654
  def Trig.sin(x)
   # ..
  end
  def Trig.cos(x)
   # ..
  end
end
```

```
module Moral
  VERY_BAD = 0
  BAD      = 1
  def Moral.sin(badness)
    # ...
  end
end
```

- Then a program to use the above modules

```
require 'trig'
require 'moral'
y = Trig.sin(Trig::PI/4)
wrongdoing = Moral.sin(Moral::VERY_BAD)
```

使用混成 (Mixin) 概念

- Ruby refuse to include the feature of multiple inheritance
  - To avoid programmer's mistakes
- Ruby's *mixin* facility is designed to simulate multiple inheritance
- A module can’t have instances, because a module isn’t a class
  - However, you can include a module within a class definition
    - Then all the module’s instance methods are suddenly available as methods in the class as well
    - All the methods in the module get mixed in
- In fact, mixed-in modules effectively behave as superclasses

混成範例

```
module Debug
  def who_am_i?
    "#{self.class.name} (\##{self.id}): #{self.to_s}"
  end
end
class Phonograph
  include Debug
  # ...
end
class EightTrack
  include Debug
  # ...
end
ph = Phonograph.new("West End Blues")
et = EightTrack.new("Surrealistic Pillow")

ph.who_am_i?   →   "Phonograph (#935520): West End Blues"
et.who_am_i?   →   "EightTrack (#935500): Surrealistic Pillow"
```

混成特性

- Mixins make it simple as just “the collection of particular properties all we want to add”

Kernel 模組

- The *Kernel* module is included by class *Object*, so its methods are available in every Ruby object
- The Kernel methods:
  - [xxx]

行為決定類型的分類法：duck-typing

- In Ruby, the class is never the type
- The type of an object is defined more by *what the object can do*
  - This is called “duck-typing”
  - *If an object walks like a duck and talks like a duck, then the interpreter is happy to treat it as if it were a duck*
- Duck-typing is not a set of rules, it's just a style of programming supported in Ruby

Duck-typing 範例

- For example, you may be writing a routine to add song information to a string. For a programmer with C# or Java background:

```
def append_song(result, song)
  # test we're given the right parameters
  unless result.kind_of?(String)
    fail TypeError.new("String expected")
  end
  unless song.kind_of?(Song)
    fail TypeError.new("Song expected")
  end

  result << song.title << " (" << song.artist << ")"
end

result = ""
append_song(result, song)   →   "I Got Rhythm (Gene Kelly)"
```

Duck-typing 範例 (2)

- Ruby is a dynamic typing language, so don't check object type like static-typed language

```
def append_song(result, song)
  result << song.title << " (" << song.artist << ")"
end

result = ""
append_song(result, song)   →   "I Got Rhythm (Gene Kelly)"
```

- Type checking is not necessary since exception will be raised if operation is not supported
  - Without the check, program will become more flexible: many types support the same operation can be passed in

本章回顧