

## 第三章     **Ruby** 的語法和語意

---

*OOP with Ruby*

# 本章內容

---

- 保留字與識別字
- 程式註解與文件
- 運算子與執行順序
- 指定表示式
- 條件分支
- 迴圈

# 保留字與識別字

- The keywords (or reserved words) in Ruby typically cannot be used for other purposes

BEGIN	END	alias	and	begin
break	case	class	def	defined
do	else	elsif	end	ensure
false	for	if	in	module
next	nil	not	or	redo
rescue	retry	return	self	super
then	true	undef	unless	until
when	while	yield		

*OOP with Ruby*

# 變數與常數

---

- Ruby variables and constants hold references to objects
- Variables themselves do not have an intrinsic type. Instead, the type of a variable is defined solely by the messages to which the object referenced by the variable responds

# 常數宣告

- A Ruby constant is also a reference to an object
- Ruby, unlike less flexible languages, lets you alter the value of a constant, although this will generate a warning message

```
MY_CONST = 1
```

```
MY_CONST = 2      # generates a warning
```

- Although constants should not be changed, you can alter the internal states of the objects they reference

```
MY_CONST = "Tim"
```

```
MY_CONST[0] = "J"      # alter string referenced by constant
```

```
MY_CONST "Jim"
```

*OOP with Ruby*

# 變數命名

- Variables and other identifiers normally start with an alphabetic letter or a special modifier. The basic rules are:
  - Local variables (and pseudo-variables such as self and nil) begin with a lowercase letter
  - Global variables begin with a dollar sign (\$)
  - Instance variables (within an object) begin with an "at" sign (@)
  - Class variables (within a class) begin with two "at" signs (@@)
  - Constants begin with capital letters
  - the underscore (\_) may be used as a lowercase letter

# 已預先定義的變數

- Ruby has several categories pre-defined variables
  - Exception Information
  - Pattern Matching Variables
  - Input/Output Variables
  - Execution Environment Variables
  - Standard Objects
  - Global Constants

# 程式註解與文件

- Comments in Ruby begin with a pound sign (#)

```
x = y + 5  # This is a comment.
```

```
# This is another comment.
```

```
print "# But this isn't."
```

- Embedded documentation is intended to be retrieved from the program text by an external tool

- Given two lines starting with =begin and =end, everything between those lines (inclusive) is ignored by the interpreter

```
=begin
```

```
The purpose of this program
```

```
is to cure cancer and instigate world peace.
```

```
=end
```

*OOP with Ruby*



# 程式區塊

- Expressions may be grouped between **begin** and **end** in Ruby

```
begin  
  body  
end
```

*OOP with Ruby*

# 運算子與執行順序

Ruby operators (high to low precedence)

Method	Operator	Description
✓	[ ] [ ]=	Element reference, element set
✓	**	Exponentiation
✓	! ~ + -	Not, complement, unary plus and minus (method names for the last two are +@ and -@)
✓	* / %	Multiply, divide, and modulo
✓	+ -	Plus and minus
✓	>> <<	Right and left shift
✓	&	“And” (bitwise for integers)
✓	^	Exclusive “or” and regular “or” (bitwise for integers)
✓	<= < > >=	Comparison operators
✓	<=> == === != =~ !~	Equality and pattern match operators (!= and !~ may not be defined as methods)
	&&	Logical “and”
		Logical “or”
	.. ...	Range (inclusive and exclusive)
	? :	Ternary if-then-else
	= %= ~= /= -= +=  = &=	Assignment
	>>= <<= *= &&=   = **=	
	defined?	Check if symbol defined
	not	Logical negation
	or and	Logical composition
	if unless while until	Expression modifiers
	begin/end	Block expression

# 指定表示式

- An assignment statement sets the variable or attribute on its left side to refer to the value on the right, It then returns that value as the result of the assignment expression

`a = 1`

`Instrument = "piano"`

`a = b = 1+2+3      # a = b = 6`

`a = (b = 1+2) + 3      # a = 6, b = 3`

# 平行指定表示式

- Ruby assignments are effectively performed in parallel

```
x = 0
```

```
a, b, c = x, (x += 1), (x += 1)    # a=0, b=1, c=2
```

```
a, b = b, a    # swapping values
```

# 其他指定表示式

---

- Shortcut expression
  - `a += 2`
  - `str += "hello"`

# 條件表示式

- Condition expressions are mostly based on boolean operators
  - AND operation
    - Both `and` and `&&` operator evaluate to true only if both operands are true. They evaluate the second operand only if the first is true
  - OR operation
    - Both `or` and `||` operator evaluate to true if either operand is true. They evaluate their second operand only if the first is false
  - NOT operation
    - Both `not` and `!` operator return the opposite of their operand (false if the operand is true, and true if the operand is false)

## 條件表示式 (2)

- The `defined?` operator returns `nil` if its argument (which can be an arbitrary expression) is not defined; otherwise it returns a description of that argument

```
defined? 1      # "expression"
```

```
defined? dummy  # nil
```

```
defined? printf  #"method"
```

```
defined? String  # "constant"
```

```
defined? $_      #"global-variable"
```

```
defined? Math::PI  # "constant"
```

```
defined? a = 1    # "assignment"
```

```
defined? 42.abs   # "method"
```

# 支援比較的方法

- In addition to the boolean operators, Ruby objects support comparison using the methods `==`, `===`, `<=>`, `=~`, `eql?`, and `equal?`

Common comparison operators

Operator	Meaning
<code>==</code>	Test for equal value.
<code>===</code>	Used to compare each of the items with the target in the when clause of a case statement.
<code>&lt;=&gt;</code>	General comparison operator. Returns <code>-1</code> , <code>0</code> , or <code>+1</code> , depending on whether its receiver is less than, equal to, or greater than its argument.
<code>&lt;</code> , <code>&lt;=</code> , <code>&gt;=</code> , <code>&gt;</code>	Comparison operators for less than, less than or equal, greater than or equal, and greater than.
<code>=~</code>	Regular expression pattern match.
<code>eql?</code>	True if the receiver and argument have both the same type and equal values. <code>1 == 1.0</code> returns true, but <code>1.eql?(1.0)</code> is false.
<code>equal?</code>	True if the receiver and argument have the same object ID.



## 支援比較的方法 (2)

---

- You can use a Ruby range as a boolean expression. A range such as *exp1..exp2* will evaluate as false until *exp1* becomes true. The range will then evaluate as true until *exp2* becomes true.

## 邏輯表示式的值

- Boolean expression in Ruby can take nil value as false

```
nil    and  true    →  nil
false  and  true    →  false
99     and  false   →  false
99     and  nil     →  nil
99     and  "cat"   →  "cat"

false  or   nil     →  nil
nil    or   false   →  false
99     or   false   →  99
```

# if 表示式

- An **if** expression in Ruby is pretty similar to “if” statements in other languages

```
if boolean-expression [ then | : ]  
  body  
[ elsif boolean-expression [ then | : ]  
  body , ... ]  
[ else  
  body ]  
end
```

```
if song.artist == "Gillespie" then  
  handle = "Dizzy"  
elsif song.artist == "Parker" then  
  handle = "Bird"  
else  
  handle = "unknown"  
end
```

## if 表示式 (2)

- If you lay out your **if** statements on multiple lines, you can leave off the **then** keyword

```
if song.artist == "Gillespie"  
  handle = "Dizzy"  
elsif song.artist == "Parker"  
  handle = "Bird"  
else  
  handle = "unknown"  
end
```

## if 表示式 (3)

- Code more tightly

```
if song.artist == "Gillespie" then  handle = "Dizzy"  
elsif song.artist == "Parker" then  handle = "Bird"  
else  handle = "unknown"  
end
```

```
if song.artist == "Gillespie":  handle = "Dizzy"  
elsif song.artist == "Parker":  handle = "Bird"  
else  handle = "unknown"  
end
```

*OOP with Ruby*

## if 表示式 (4)

- Ruby's `if` expression returns a value

```
handle = if song.artist == "Gillespie" then
  "Dizzy"
elsif song.artist == "Parker" then
  "Bird"
else
  "unknown"
end
```

## unless: if 相反詞

- Ruby also has a negated form of the if statement

```
unless boolean-expression [ then | : ]  
  body  
[ else  
  body ]  
end
```

```
unless song.duration > 180  
  cost = 0.25  
else  
  cost = 0.35  
end
```

*OOP with Ruby*

# C 語法式的條件表示式

---

- Ruby also supports the C-style conditional expression

```
cost = song.duration > 180 ? 0.35 : 0.25
```

*OOP with Ruby*



# 進階的條件分支表示式

- Tack conditional statements onto the end of a normal statement

```
mon, day, year = $1, $2, $3 if date =~ /(\d\d)-(\d\d)-(\d\d)/  
puts "a = #{a}" if debug  
print total unless total.zero?
```

# case 條件分支

- The first form of case allows a series of conditions to be evaluated, executing code corresponding to the first condition that is true.

```
case
when condition [, condition ]... [ then | : ]
    body
when condition [, condition ]... [ then | : ]
    body
...
[ else
    body ]
end
```

## case 條件分支 (2)

- Ruby's **case** expression returns a value

```
leap = case
  when year % 400 == 0: true
  when year % 100 == 0: false
  else year % 4 == 0
end
```

## case 條件分支 (3)

- The second form of a case expression takes a target expression following the case keyword
  - It searches for a match by starting at the first (top left) comparison, performing `comparison === target`

```
case target
when comparison [, comparison ]... [ then | : ]
  body
when comparison [, comparison ]... [ then | : ]
  body
...
[ else
  body ]
end
```

## case 條件分支 (4)

- You specify a target at the top of the **case** statement, and each **when** clause lists one or more comparisons

```
case input_line
when "debug"
  dump_debug_info
  dump_symbols
when /p\s+(\w+)/
  dump_variable($1)
when "quit", "exit"
  exit
else
  print "Illegal command: #{input_line}"
end
```

## case 配合 range 使用

```
kind = case year
  when 1850..1889 then "Blues"
  when 1890..1909 then "Ragtime"
  when 1910..1929 then "New Orleans Jazz"
  when 1930..1939 then "Swing"
  when 1940..1950 then "Bebop"
  else
    "Jazz"
  end
```

```
kind = case year
  when 1850..1889: "Blues"
  when 1890..1909: "Ragtime"
  when 1910..1929: "New Orleans Jazz"
  when 1930..1939: "Swing"
  when 1940..1950: "Bebop"
  else
    "Jazz"
  end
```

# 迴圈概述

- Ruby uses several loop control keywords
  - while
  - until
  - each
  - loop method
- Ruby doesn't have a “for” loop—at least not the kind you'd find in C, C++, and Java
  - Ruby uses methods defined in various built-in classes to provide equivalent, but less error-prone, functionality.

## 簡易迴圈範例

- Use `times`, `upto`, `step` method

```
3.times do
  print "Ho! "
end
```

Ho! Ho! Ho!

```
0.upto(9) do |x|
  print x, " "
end
```

0 1 2 3 4 5 6 7 8 9

```
0.step(12, 3) {|x| print x, " " }
```

0 3 6 9 12

*OOP with Ruby*



## 迴圈 : while

- while and until loops are the most common used in Ruby
- **while** executes *body* zero or more times as long as boolean-expression is true

```
while boolean-expression [ do | : ]  
  body  
end
```

- Another form

```
expression while boolean-expression
```

## 迴圈 : until

- **until** executes *body* zero or more times as long as boolean-expression is false

```
until boolean-expression [ do | : ]  
  body  
end
```

- Another form

```
expression until boolean-expression
```

## 迴圈 : each iterator

- The for loop is executed as if it were the following **each** loop

```
expression.each do | name [, name ]... |  
  body  
end
```

```
[ 1, 1, 2, 3, 5 ].each {|val| print val, " " }
```

```
1 1 2 3 5
```

## 迴圈 : loop

- `loop`, which iterates its associated block, is not a language construct—it is a method in module Kernel

```
loop do
  print "Input: "
  break unless line = gets
  process(line)
end
```

# for 迴圈 ?

- **for** is almost a lump of syntactic sugar

- When you write

```
for song in songlist  
  song.play  
end
```

- Ruby will translate into

```
songlist.each do |song|  
  song.play  
end
```

*OOP with Ruby*

## 另一個 for 範例

```
for i in ['fee', 'fi', 'fo', 'fum']  
  print i, " "  
end  
for i in 1..3  
  print i, " "  
end  
for i in File.open("ordinal").find_all {|line| line =~ /d$/}  
  print i.chomp, " "  
end
```

```
fee fi fo fum 1 2 3 second third
```

# 再談程式區塊

- As mentioned earlier, expressions may be grouped between **begin** and **end** in Ruby

```
begin
  body
end
```

- However, we can also use braces ({} ) to declare a C-style block:

```
presidents = ["Ford", "Carter", "Reagan", "Bush1", "Clinton", "Bush2"]
presidents.each {
  |prez|
  puts prez
}
```

```
presidents = ["Ford", "Carter", "Reagan", "Bush1", "Clinton", "Bush2"]
presidents.each do
  |prez|
  puts prez
end
```

## 再談程式區塊 (2)

- One small difference between brace enclosed blocks and do/end enclosed blocks

```
my_array = ["alpha", "beta", "gamma"]
puts my_array.collect {
  |word|
  word.capitalize
}
puts "====="
puts my_array.collect do
  |word|
  word.capitalize
end

Alpha
Beta
Gamma
=====
alpha
beta
gamma
```

*with Ruby*



## 再談程式區塊 (3)

---

- Generally speaking, if you want to directly use the result of iterators, use braces
- For longer blocks, do/end is more readable, and the overhead for the extra variable and line of code is trivial

# 使用 **break**, **redo**, **next**, **retry** 變更流程

- **break** terminates the immediately enclosing loop—control resumes at the statement following the block
- **redo** repeats the loop from the start, but without reevaluating the condition or fetching the next element
- **next** skips to the end of the loop, effectively starting the next iteration
- **retry** restarts the loop, reevaluating the condition

## break/next 範例

```
i=0
loop do
  i += 1
  next if i < 3
  print i
  break if i > 4
end
```

345

## retry 範例

```
for i in 1..100
  print "Now at #{i}. Restart? "
  retry if gets =~ /^y/i
end
```

```
Now at 1. Restart? n
Now at 2. Restart? y
Now at 1. Restart? n
```

# 本章回顧

---

*OOP with Ruby*