## Control Flow

### return statement and methods

The return statement can be used to return from a method. The return statement is *optional* in Ruby.

def add(x, y)  
 x + y  
end  
def add\_return(x, y)  
 return x + y  
end  
add(2, 2)  
=> 4  
add\_return(2, 2)  
=> 4

In this example, the result of the last expression is returned.

The return statement is used when multiple return values are desired:

def reverse(x, y)  
 return y, x  
end  
reverse(2, 4)  
=> [4, 2]

Notice that the result is returned as an array, and can be used in conjunction with parallel assignment:

a, b = reverse(2, 4)  
puts a, b  
4  
2

The return statement is used when you want to exit a method prematurely:

def fact(n)  
 return 1 if n < 2  
 return n \* fact(n – 1)  
end

### return statement and blocks/procs

The return statement returns from the lexically enclosing method:

class Array  
 def element\_greater\_than(max)  
 self.each { |e| return true if e > max }  
 false  
 end  
end  
[1,2,3,4,5].element\_greater\_than(4)  
=> true  
[1,2,3].element\_greater\_than(3)  
=> false

Notice that the return statement returns from the element\_greater\_than method, and not from the block.

The return statement can be implemented as a ret statement with a distinguished return value (BlockReturnReason).

Since the semantics of yield and Proc#call are identical, I will document the remainder of the behavior of return in terms of procs only:

def create\_and\_call  
 p = Proc.new { return }  
 **p.call**  
 puts 'never executed'  
end  
  
create\_and\_call

def yield\_to\_block  
 yield  
end  
  
def create\_and\_yield  
 p = Proc.new { return }  
 **yield\_to\_block &p # convert to block** puts 'never executed'  
end  
  
create\_and\_yield

The return statement will return from arbitrarily deeply nested procs.

def call\_nested\_proc  
 p = Proc.new { |o| o.call }  
 n = Proc.new { return }  
 p.call n  
 puts 'never executed'  
end  
  
call\_nested\_proc

An eval'd return statement behaves the same as an inline return statement.

def call\_nested\_proc  
 p = Proc.new { |o| o.call }  
 n = Proc.new { **eval('return')** }  
 p.call n  
 puts 'never executed'  
end  
  
call\_nested\_proc

Ruby will raise a LocalJumpError if you attempt to return to a place on the call stack that no longer exists:

def yield\_to\_block  
 yield  
end

def create\_proc  
 Proc.new { return }  
end  
  
yield\_to\_block &create\_proc  
=> LocalJumpError

Ruby will raise a ThreadError if you attempt to return across a thread boundary:

t = Thread.new { return }  
t.join

### Return statement and lambdas

The return statement behaves differently with lambdas; it returns from the block to the caller.

def create\_and\_call  
 p = lambda { return }  
 p.call  
 puts 'reached'  
end  
  
create\_and\_call

The return statement always returns from a block, regardless of nesting.

def create\_and\_call  
 p = lambda do |o|   
 o.call  
 puts 'reached'  
 end  
 n = lambda { return }  
 p.call n  
 puts 'reached'   
end  
  
create\_and\_call

### break statement and methods

In a loop, the break statement transfers control to the expression following the loop:

i = 0  
while i < 10 do  
 break if i == 5  
 i += 1  
end  
puts i  
=> 5

The break statement can also return a value:

i = 0  
**x =** while i < 10 do  
 **break i** if i == 5  
 i += 1  
end  
**puts x**=> 5

The break statement can return multiple values as an array:

i = 0  
**x, y =** while i < 10 do  
 **break 2, 3** if i == 5  
 i += 1  
end  
**puts x, y**2  
3

The break statement transfers control to the expression following the iterator method:

def yield\_to\_block  
 yield  
 puts 'never executed'  
end  
  
def create\_and\_yield  
 **x =** yield\_to\_block { break 42 }  
 puts "reached #{x}"  
end  
  
create\_and\_yield

Note that this behavior is different than return, which returns from the create\_and\_yield method.

In the case where the yields are nested, it returns only one level:

def yield\_block  
 yield  
end  
  
yield\_block {  
 yield\_block {  
 break  
 }  
 puts 'reached'  
}  
puts 'reached'

Ruby will raise a LocalJumpError if you attempt to return to a place on the call stack that no longer exists:

def yield\_block  
 yield  
end  
  
def create\_proc  
 Proc.new { break }  
end  
  
yield\_block &create\_proc