# Block, Proc, Lambda

---

## Block

If you have used each before, then you have used blocks!

A code block is a chunk of code you can pass to a method, as if the code block were another parameter.

A code block is a set of Ruby statements and expressions inside braces or a do/end pair.

The block may start with an argument list between vertical bars.

A code block may appear only immediately after a method invocation, and the start of the block (the brace or the do) must be on the same logical source line as the end of the invocation.

### Why do block exist?

A block is useful because it allows you to save a bit of logic (code) & use it later.

### Is a block of the "type" Proc?

In Ruby, a block is conceptually related to a Proc object, although they are not exactly the same.

Blocks are not objects themselves, but they can be associated with objects like Proc and Lambda.

A Proc (short for *procedure*) object is an instance of the Proc class, which encapsulates a block so that it can be stored in a variable, passed to a method, or manipulated in other ways that blocks alone do not allow. A Proc can be thought of as a "block object." A Proc object can be called just like a method and will execute the block of code it contains.

When you use a block in a method call, like this:

```
[1, 2, 3].each { |x| puts x }
```

You are passing a block directly to the each method. This block is not an object and cannot be manipulated as one.

However, when you create a Proc object, you are turning a block into an object that can be stored, passed around, or called. For example:

```
my_proc = Proc.new { |x| puts x }
[1, 2, 3].each(&my_proc)
```

In this case, my_proc is a Proc object that encapsulates the block. The & operator is used to convert the Proc object back into a block when passing it to the each method.

In summary, while a block itself is not an object, a Proc is an object type that represents a block in Ruby, allowing for more flexibility in how the block is used.

Blocks can be “explicit” or “implicit”. Explicit means that you give it a name in the parameter list. You can pass an explicit block to another method or save it into a variable to use later.

```
def broadcast( &work )
     work.call # same as yield
end
broadcast { puts "Explicit block called" }
```

Notice the &work paramete. That’s how you define the block’s name!

{...} is a block. do ... end is a block

While blocks are a key part of Ruby, they are not objects until converted into a Proc or Lambda.

### yield

yield is a keyword that calls a block when you use it. It’s how methods **USE** blocks!

When you use the yield keyword, the code inside the block will run to do its job, just like when you call a regular method.

---

## Lambda

A lambda is a way to define a block and its parameters with some special syntax. You can save this lambda into a variable for later use.

A **Lambda** is created with lambda {} or -> {}.

```ruby
say_something = -> { puts "This is a lambda." }
# Or construct a proc with lambda semantics using the Kernel#lambda method.
talk_anything = lambda { puts "This is also a lambda." }
```

Defining a lambda won't run the code inside it, just like defining a method won't run the method, you need to use the `call` method for that.

### What is a lambda in general?

In general programming concepts, a lambda is a type of anonymous function. The term originates from lambda calculus, a formal system in mathematical logic.

1. **Anonymous:** Lambda functions are usually anonymous, meaning they don't have a name like traditional functions. This makes them convenient for short, throwaway functions that are used only in a particular context.
2. **Concise Syntax:** Lambdas typically have a more concise syntax compared to regular functions, making them easier to read and write for small functionalities.
3. **Immediate Execution:** Unlike regular functions, lambdas can be executed immediately at the place where they are defined.
4. **Closure:** Lambdas often form closures, meaning they can capture and use variables from their enclosing scope. This makes them powerful for various programming patterns, such as callbacks, event handlers, and short-lived operations that require some local state.

---

## Proc

A **Proc** is created with `proc {}` or `Proc.new {}`.

Proc objects are *closures*, meaning they remember and can use the entire context in which they were created.

### The difference between a Proc and a Lambda

In Ruby, both Procs and Lambdas are objects belonging to the `Proc` class, and they are used to encapsulate blocks of code.

**Lambdas** are closer to a "function" in the traditional sense. They are expected to behave like methods with respect to arguments and return values. **Procs** are more like "blocks of code" that can be executed.

Lambdas are generally preferred when you need function-like behavior, while Procs are useful for more flexible code blocks where such strictness is not required.

**Argument Handling:**

- Lambdas check the number of arguments passed to them and will throw an error if the number of arguments does not match the expected number.
- Procs are more flexible with arguments. If you pass the wrong number of arguments to a Proc, it won't complain. If it's missing arguments, they will be set to nil.

**Return Behavior:**

- When you return from a Lambda, it returns control to the enclosing method, just like a regular method return.
- When you return from a Proc, it does so immediately, without going back to the calling method. This is known as a non-local return.

These differences can lead to different behaviors in code.

**Semantic Differences:** Ruby differentiates between Proc and Lambda for subtle but important reasons related to how they handle arguments and return values. This distinction allows programmers to choose the behavior that best suits their needs, which would be less clear with a generic Function class.

---

## Are the purpose of existing for a Proc or Lambda just for encapsulating a block?

The primary purpose of Proc and Lambda in Ruby is indeed to encapsulate blocks of code. These features allow for more dynamic and flexible programming, characteristic of Ruby's approach to software design.

The purposes and uses of Procedure and Lambda are:

1. **Encapsulation of Code for Reuse:** Both Proc and Lambda allow you to store blocks of code in variables, pass them as arguments to methods, and call them multiple times. This encapsulation facilitates code reuse.
2. **Delaying Execution:** They enable you to write a block of code and run it at a later time, which is essential for callbacks, deferred execution, and implementing things like iterators or asynchronous code.
3. **Functional Programming:** They allow Ruby to support functional programming concepts, where functions (in Ruby's case, Procs and Lambdas) can be passed around and manipulated just like any other object.
4. **Context Preservation:** When you create a Proc or Lambda, it captures the current execution context (variables, methods, etc.). This feature, known as closure, allows the code inside the Proc or Lambda to access those external variables and methods even if it's executed in a different context.

5. **Control Flow:** Lambdas can also influence control flow with `return`, `break`, and `next`. A `return` from a lambda returns from the lambda itself, while a `return` from a `Proc` returns from the enclosing method, which can be useful for certain control flow patterns.
6. **Behavioral Differences:** Lambdas check the number of arguments passed to them and will raise an error if the number of arguments doesn't match. Procs, on the other hand, are more flexible and simply assign `nil` to any missing arguments.
7. **Custom Iterators and Enumerable Methods:** They are used to build custom iterators and influence the behavior of enumerable methods (`each`, `map`, `select`, etc.) by providing specific actions to perform on each element of a collection.

In summary, while the encapsulation and reuse of blocks of code are primary purposes of `Proc` and `Lambda`, their roles in Ruby encompass enabling delayed execution, supporting functional programming patterns, capturing context, controlling flow, and customizing behavior in enumerable operations. This flexibility makes them powerful tools in the Ruby programmer's toolkit.

---

## Why doesn't Ruby just use a Function concept over a Proc and a Lambda?

A lambda 是傳統、嚴格意義上的 function。不知道，Ruby為何不直接稱之為 function。

Ruby's use of `Proc` and `Lambda` instead of a distinct `Function` class is a reflection of its design philosophy and the influences from other programming languages. Understanding the reasons requires a look at Ruby's roots and design goals:

1. **Everything is an Object:** Ruby follows the principle that everything should be an object. This principle is fundamental to its design. Unlike some languages where functions are a separate construct, in Ruby, even these "callable objects" are just that: objects. This approach is consistent with Ruby's object-oriented nature.
2. **Influence from Lisp and Smalltalk:** Ruby's design is heavily influenced by Lisp and Smalltalk, both of which treat code blocks as first-class citizens that can be passed around like objects. This influence led to the adoption of `Proc` and `Lambda` as objects representing blocks of code, rather than having a separate `Function` class.
3. **Flexibility and Simplicity:** Ruby emphasizes simplicity and productivity. Having a unified way to handle blocks of code (through `Proc` and `Lambda`) simplifies the language's model. You don't need to learn different syntax or rules for functions versus other objects; everything follows the same object-oriented principles.
4. **Closure Support:** Both `Proc` and `Lambda` are closures, meaning they can capture and carry the context (variables, bindings) in which they are defined. This capability is central to many Ruby idioms and patterns, especially in functional programming style within an object-oriented language.

5. **Consistency with Blocks:** In Ruby, blocks are anonymous pieces of code used throughout the language (in iterators, for instance). `Proc` and `Lambda` provide a seamless way to work with these blocks as objects, maintaining consistency in the language's design.

In summary, the use of `Proc` and `Lambda` in Ruby over a distinct `Function` class is a deliberate design choice that aligns with the language's object-oriented nature and its emphasis on simplicity, flexibility, and consistency. This approach provides a Ruby-like way to handle functions as first-class objects while supporting the language's functional programming capabilities within an object-oriented context.

## Why doesn't Ruby just name the class Block or Function, instead of Proc and Lambda?

Ruby's choice to use the terms "Proc" and "Lambda" instead of "Block" or "Function" is deeply rooted in its design philosophy and historical influences. Understanding these choices involves looking at both the linguistic choices and the technical distinctions Ruby aims to make:

**Historical Influences and Terminology:**

- **Proc:** The term "Proc" is short for "procedure," a concept borrowed from earlier programming languages. In programming language theory, a procedure is a set of instructions that perform a task. This term is used in many languages, including Lisp, from which Ruby draws significant influence. By using "Proc," Ruby aligns with a long-standing tradition in programming languages.
- **Lambda:** The term "Lambda" comes from the lambda calculus, a formal system in mathematical logic and computer science for expressing computation based on function abstraction and application. Lambda calculus has greatly influenced functional programming languages. In many of these languages, "lambda" refers to anonymous functions, which is consistent with how Ruby uses the term.

**Conceptual Clarity:**

- **Why Not "Block"?:** In Ruby, a block is an anonymous piece of code that can be passed to methods, but it is not an object itself. Using "Block" for what Ruby calls a "Proc" could be misleading because blocks and Procs, while related, are not the same. A block becomes a Proc when it is objectified, but they have different behaviors and properties.
- **Why Not "Function"?:** In many languages, "function" refers to named, standalone entities in the code. Ruby methods are more akin to what other languages call functions. Using "Function" for Procs or Lambdas could cause confusion, especially given that Procs and Lambdas in Ruby have distinct behaviors from regular methods (like differing arity rules and return behaviors).

In summary, Ruby's choice to use "Proc" and "Lambda" is a reflection of its historical influences, desire for conceptual clarity, consistency with its own design principles, and the need to distinguish between different behaviors of these callable objects. These terms provide both a nod to programming history and a clear indication of their roles and behaviors within Ruby.

---

# The Ampersand & Operator

`[2, 4, 6].select!( &:even? )` is equivalent to `[2, 4, 6].select! { |i| i.even? }`

The `&:even?` is shorthand for a block `{ |i| i.even? }`.

`:even?` is a method that returns `true` if an integer is even and `false` otherwise. The colon `:` before `even?` indicates that it is a symbol, namely, the `:` before `even?` converts the name of the method into a symbol. So, `:even?` represents the symbol form of method `even?`.

The `&` operator is used to convert the symbol `:even?` into a Proc object, which is then passed to the `select!` method. It will allow us to call the method named `even?` on each element of the array.

Since `!` is used after `select`, it modifies the original array itself.

The `&` operator in Ruby is quite versatile and serves different purposes depending on the context, particularly when dealing with blocks, Procs, and methods.

## 1. Converting Blocks to Procs and Vice Versa

**Turning a Block into a Proc:** When defining a method, if you prefix the last parameter with `&`, any block passed to this method is converted to a `Proc` object. This allows you to store the block in a variable, pass it around, or call it later.

```ruby
def my_method &block
     block.call
end
my_method { puts "Hello, world!" } # => "Hello, world!"
```

The method is defined with an argument named `block`, prefixed with `&`. This means that if a block is given when the method is called, it will be converted to a `Proc` object and assigned to the parameter `block`.

% 當 my_method 被調用時，block 才被賦予了 "Hello, world!" 這個值。 20230113 %

When a method is defined with a parameter using the & syntax, it means that the method can accept a block, but it's not mandatory to pass one. However, if you don't pass a block and then try to call the block variable as a Proc object (with block.call), an error will be encountered because block will be nil if no block is provided. To handle this case, you should check if the block is nil before calling it.

```
def my_method(&block)
     block.call if block
end
```

**Passing a Proc as a Block:** When calling a method that expects a block, you can pass a Proc object with an & prefix. This converts the Proc back into a block.

```
my_proc = proc { puts "Hello from Proc!" }
some_method(&my_proc)
```

Here, some_method is called with my_proc being converted to a block.

## 2. Converting Method Objects to Blocks

**Referencing a Method:** You can reference a method on an object using the method method, which returns a Method object. You can then convert this Method object to a block by using the & operator. This is useful for passing a method as a block to another method.

```
class MyClass
     def my_method
          puts "Method called"
     end
end

obj = MyClass.new
method_obj = obj.method(:my_method)
[1, 2, 3].each(&method_obj)
```

In this example, method_obj (a Method object) is converted to a block and passed to each.

## 3. Symbol to Proc Conversion

**Shorthand for Calling Methods:** The & operator is also used in a popular Ruby idiom that converts symbols to blocks, known as the "Symbol to Proc" conversion. This is often used with methods like map, select, or each.

```
[1, 2, 3].map &:to_s
```

In this case, `&:to_s` converts the `:to_s` symbol to a `Proc` that calls the `to_s` method on every object in the array.

## Summary

The `&` operator in Ruby is a powerful tool, used for converting blocks to Procs and vice versa, converting method references to blocks, and enabling the concise Symbol to Proc idiom. This operator is a key part of Ruby's flexible and expressive method and block handling capabilities, contributing to the language's reputation for elegant and readable code.