Ruby Explained

Attribute accessor

This method is used to expose instance variables (which otherwise can't be accessed from outside the instance) by generating "accessor" methods. Two simple methods are generated; one for reading that returns the instance variable of the same name, the other for writing that assigns a new value to the instance variable. For instance, if attr_accessor :programming_language was used in a "Person" class, it would expose the attribute on all its instances:

```
person.programming_language
#=> nil

person.programming_language = "Ruby"
person.programming_language
#=> "Ruby"

# attribute value is stored in "@programming_language"
# instance variable on the "person" object
```

Attribute reader

This method is used to expose instance variables (which otherwise can't be accessed from outside the instance) by generating a "reader" method. The generated method simply returns the value of an instance variable of the same name.

Attribute writer

This method is used to easily enable writing to instance variables (which otherwise can't be accessed from outside the instance) by generating a "writer" method. The generated method takes a single argument and writes the given value to the instance variable of the same name.

begin statement

A begin statement is primarily used when a piece of code may raise an exception that we wish to rescue from or ensure that even if an exception is raised that more code is ran. An example:

```
begin
  boom!
rescue Boom => e
  puts "No more boom."
ensure
  puts "Everything is OK."
end
```

In this example, if the boom! method raises a Boom exception, this will be caught by the rescue statement which assigns the rescued exception to the local variable e. Everything after the rescue but before the following end or ensure is the code that will be ran during the rescue. To get the stacktrace or message of this exception we can call stacktrace and message respectively on this object.

The ensure statement declares code that should be ran always, regardless of if an exception was rescued.

case statement

A case statement is used for when you want to act on a returned value that could be one of many different values. The value passed to the case statement is the value which is being checked. The when statements inside the case use a triple-equal (===) call to determine if the two objects are equal, using the value from the when as the left-hand side of the operation and the value from the case as the right-hand side.

This triple-equal sign is how you may check to see if the specified object is of a specific type:

```
>> String === "a"
=> true
```

Class definition

The class keyword defines a named class. Classes are objects which group together methods that will be properties of objects instantiated from the class. Every object in Ruby is an instance of a class; e.g. numbers are instances of the "Numeric" class and strings of the "String" class. The same way they are created, classes can be reopened and new methods can be added to them; when that happens, newly added methods are instantly available on all existing objects of that class.

Class initializer

The "initialize" method, if defined, is the method that always gets invoked automatically on each new instance of the class. It is used to initialize instance variables and process given arguments, if any. For example, if a "person" object is instantiated with Person.new("Ruby"), the "initialize" method would get called with a single argument with a value of "Ruby". By default, all arguments that new was called with are forwarded to the "initialize" method of the new instance.

Singleton class access

This construct is a way to enter the scope of the current singleton class, otherwise known as "metaclass". The concept of a singleton class is somewhat advanced; accessing it is often referred to as "metaprogramming". In simple terms, every class and module in Ruby has its singleton class, and defining instance methods on the singleton class actually defines *class* methods on the corresponding class/module. The class << self construct is most commonly used to define class methods without having to use the "self." prefix which would otherwise be required.

Class method

A *class method*, denoted by the "self." prefix on its name, is a method that can be called directly on a class or module without needing to create an instance of that class. This is opposed to regular *instance methods* which are only available on instances of a class. Class methods are used to implement behavior that this class is responsible for, but which isn't tied to any particular instance.

Class inheritance

A class can inherit from another class; this is denoted by the `<` sign. The new class becomes a "subclass" of its parent "superclass" and inherits all methods from the parent class. In the new class, new methods can be added and inherited methods overridden; nothing of this will affect the superclass.

:: separator

A `::` separator is used to indicate that the right-hand-side constant is defined within the scope of the left-hand-side constant. Take this for example:

```
module Foo
  class Bar

  end
end
```

To get to the `Bar` class we would use `::` like this: `Foo::Bar`.

:: prefix

When a constant is prefixed with `::` it means that Ruby should look for this constant at the root level.

Constant

A *constant* is a pointer to an object, just like a variable. Their names start with an uppercase letter, and they are often all in uppercase. The difference from a variable is that a constant shouldn't be reassigned to another object; Ruby will allow it but issue an "already initialized constant" warning. Constants are usually used to hold bits of hardcoded data used in a namespace, e.g. `HTTPVersion = "1.1"` can be found in "Net::HTTP" and is used internally when communicating over HTTP.

Another thing that might not be immediately obvious is that names of classes and modules are constants, too; they start with an uppercase letter. These are regular constants which point to an object that represents the class or module. For instance, there are three constants in the expression "Net::HTTP::HTTPVersion": the first one points to the "Net" module, the second one to the "HTTP" class inside the first module, and the third one points to the string "1.1" (as mentioned above).

defined?

The `defined?` method is used to check if a particular variable, constant or object is defined or set. This will return a string representation of how this is defined. For example:

```
>> a = 1
>> defined?(a)
=> "local-variable"
```

for statement

A for statement is used for iterating over a collection of objects. The code between the for statement and its matching end will be ran for every element in the specified collection, defining a local variable for every object for each time the block is ran.

if/unless statement

The if statement is used for when we want to conditionally run code.

unless is the opposite of if. Where if will run the code if it evaluates to something that is not nil or false, unless will run its contained code *only* if the condition evaluates to nil or false.

A common if syntax is this:

```
if condition then
  conditional_code
end
```

If condition in this example is not nil or false then the conditional_code will be executed. The then after the condition can be implicit and doesn't need to be there. Because this if is short, it can be all put on to one line:

```
conditional_code if condition
```

You may also use the else keyword after the if but before the end to define code that should be ran if the if condition evaluates to nil or false:

```
if condition
  conditional_code
else
  alternative_code
end
```

A shorter way to write this would be to use a *ternary statement*, which looks like this:

```
condition ? conditional_code : alternative_code
```

In this example the ? indicates the beginning of the code to run when the if evaluates to true. After that code, we use : to denote what code should be ran if the condition evaluates to false.

Please note: if statements that can be condensed to one line will be output as ternary examples, where as multi-lined if statements will be output as such with then after the condition.

Module include

The include method allows a module to be included in a class or another module. The module being included is commonly referred to as "mixin", because its methods are being *mixed-in* with existing methods in the current class/module. Multiple modules can be included in the same object, and even if new methods are defined in the included model afterwards, these methods will immediately be available in every object where this module was mixed-in.

String Interpolation

When using double quoted (") strings, you can dynamically add more content by using a process called interpolation. Any content inside a double quoted string wrapped in #{} will be evaluated and the output added to the string which contains it.

Method definition

A method is a function defined and available on an object. A method is a block of code that can receive zero or more arguments, execute and return a value. Most commonly, methods are defined inside classes and modules; they define properties that all instances of the same class share. The code inside a method has access to *instance variables* of the current object.

Unless an explicit return statement is used, all methods return the value of their last expression when they run. If the method is defined with a list of arguments, the caller has to pass the required number of arguments when calling the method.

"Bang" method

A method that ends with an exclamation point (!) is otherwise known as a "bang" method. These methods are like any other methods, but since the exclamation point distinguishes them from other method calls, they are are most commonly used for destructive operations that either change the receiving object or trigger an irreversible operation. Sometimes a regular method will have its "bang" counterpart; for instance a lot methods on Array objects have counterparts with an exclamation point in the name, indicating the second method *changes* the array object instead of returning a new one; e.g. array.reverse returns a new array which is the reversed copy of the original, while array.reverse! changes the original array without making a copy.

Block method argument

Every method in Ruby can take a "block", and this method indicates it explicitly by declaring a named block argument. This special argument is denoted by the & sign before its name, must come at the last place in the arguments list, and can only be one per method. The "block" argument is always optional; if not given it will be nil.

A "block" in Ruby is simply some Ruby code grouped together, delimited by do ... end keywords or, alternatively, curly brackets ({ ... }) following a method call. Like methods, blocks can receive arguments; the argument list is specified with pipe symbols (e.g. |a, b, c|) at the beginning of the block.

At execution time, the method can check if it received the block. This is done with a special language construct block_given?. The block can be invoked with its call method, e.g. block.call(1, 2, 3). The values supplied will be values for block arguments, if any.

Default argument values

A method can be defined with default argument values, denoted by assignment syntax in the list of arguments. Arguments with default values must come *after* regular arguments and are commonly referred to as "optional arguments".

"Predicate" method

A method that ends with a question mark (?) is otherwise known as a "predicate" method. These methods are like any other methods, but since they look like questions they are commonly implemented so that they return true or false. This is useful in conditional expressions, e.g. if size.zero? and file.closed?.

"Setter" method

A method that ends with an equal sign (=) is otherwise known as a "setter" method. Defining setter methods (or using attr_writer/attr_accessor to generate them) is the only way to enable assignment-like syntax for properties of the class. A setter method must take an argument. For example, the setter method "name=" on a "Person" class can be invoked in two styles: person.name=("Ruby") or person.name = "Ruby". The syntaxes both perform the same function but the second one is preferred since it looks like regular variable assignment.

Module definition

A module groups together method definitions, constants, classes and other modules. Modules can be *included* by classes and other modules to import the original module's methods; this behaves similar to class inheritance but is not limited to a single include. Modules are also often used to group together other classes and modules which effectively provides a *namespace* for them; for instance "Net" is a module from Ruby standard library that serves as a namespace for Net::HTTP and related classes.

Range definition

By using .. between two values, Ruby will define an *inclusive* Range object. That is to say that if this is done:

```
"a".."z"
```

All letters from a through z will be included in this Range object. If you only want everything until the last value, use ... like this:

```
"a"..."z"
```

This will return the letters a-y.

require method

The require method loads external code. It scans the $LOAD_PATH array of filesystem paths and loads the first matching file found in one of the paths. The method will never load a ruby script on the same path more than once, and the ".rb" file extension is usually omitted.

By default, the "load path" array includes the standard library of the current ruby installation, but the require method can also be used to load 3rd-party libraries through package managers such as RubyGems.

Call the overridden method implementation

The super keyword is a way of calling the method implementation that was overridden by the current implementation. It is typically used in a subclass to invoke the same-named method defined in the superclass.

=~ operator

The =~ operator is used when we wish to compare a string to a regular expression. The string and regular expression may be placed on either side of this operator, Ruby does not care.

If the regular expression matches the string then this operator returns the digit representing the character position of the first match, beginning from 0. If there is no match then nil is returned.

Class variable

A class variable is a pointer to an object and is global inside a class, its subclasses and all their instances. They start with a double @@ character. These variables are used to carry class configuration that is shared both in its instances and the class itself.

Instance variable

An instance variable is a pointer to an object and is available only inside a single object (an "instance"). They start with a single @ character. These variables are used to carry object state, which differentiates one instance from another. An instance variable can be accessed before it is even assigned; its default value will be nil. However, in verbose mode the ruby interpreter will issue a warning about accessing a variable that is not declared. Instance variables are often initialized in the "initialize" method in a class.

Local variable

A local variable is a pointer to an object and is available only in the local scope. Variables don't have to be declared upfront and can hold values of any type.