Thread-safe Singleton

This lesson discusses correctly implementing a singleton pattern in Ruby.

Thread-safe Singleton

You are designing a library of superheroes for a video game that your fellow developers will consume. Your library should always create a single instance of any of the superheroes and return the same instance to all the requesting consumers.

Say, you start with the class `Superman`. Your task is to make sure that other developers using your class can never instantiate multiple copies of superman. After all, there is only one superman!

Solution

You probably guessed we are going to use the **singleton** pattern to solve this problem. The singleton pattern sounds very naive and simple but can be tricky to implement correctly especially in a multi-threaded environment.

First let us understand what the pattern is. **A singleton pattern allows only a single object/instance of a class to ever exist during an application run.** Typical uses of singleton pattern include creating objects that represent configuration data, global logging systems, and other similar structures.

Following are some of the ways of implementing the Singleton design pattern in Ruby:

Using Singleton Module

The first implementation makes use of the `Singleton` module offered by Ruby's standard library. The only difference between a module and a class in Ruby is that a module cannot be instantiated. A significant benefit of using the `Singleton` module is that it uses lazy instantiation i.e the instance is created only when its required and not at class loading time. When the `Singleton` module is included in a class then the `new()` method for that class becomes private for scope outside the class and is only accessible by methods within the class.

In this example, we have a class named `Superman` consisting of only one method `fly()` that prints a fly message.

```
require 'singleton'

class Superman
    include Singleton

        def fly
                        puts "Fly!"
        end
end

Superman.instance().fly
```

The standard library's singleton module does a lot of heavy lifting under the hood:

- Overrides `inherited()` on the class to ensure that subclasses also retain Singleton behavior.
- Overrides `dup()`/`clone()` on the class to ensure that copied classes also retain Singleton behavior.
- Overrides `_load()` to call instance(), modifying Marshal loading behavior to return the single instance.
- Overrides `_dump()` to strip state information when serializing via Marshal
- Overrides `dup()`/`clone()` on the instance to raise a TypeError, preventing cloning or duping of the instance

Using Class Method and Class Variables

We can use class method and variables to define a singleton too. There are two requirements to make a class adhere to the singleton pattern when using class methods and variables:

- First, hide the `new` method in the scope outside the `Superman` class. We can use the macro `private_class_method` which marks the `new` method private and inaccessible outside the `Superman` class.
- The second part of the trick is to create a class/static method, usually named `getInstance()` that returns the only instance of the class.

In the example below, we have created a class variable `instance` that points to the only `Superman` we create. Five threads invoke the `fly()` method all at once on the singleton object.

```
class Superman

    def fly
          puts "Superman object #{@@instance} is Flying!"
        end

        def self.instance
          @@instance
    end

    @@instance = Superman.new

    private_class_method :new
end

threads = Array.new

5.times do
    threads << Thread.new do
         Superman.instance().fly
    end
end

threads.each(&:join)
```

Thread-safety isn't a concern because the singleton instance is initialized inline. Note that in the above example, we instantiate the singleton `Superman` object even if the application never uses it. This might be a problem if initializing the singleton is an expensive operation and there's no guarantee if it will be used in a given run of the application. This approach of instantiating the singleton when the application loads classes, is called eager initialization.

Lazy Initialization

Lazy initialization refers to the approach of intializing an object when it is requested for the first time. The initialized object is returned for all subsequent requests. The previous example can be refactored as a lazy initialization implementation as follows:

```
class Superman
```

```
        def fly
        puts "Superman object #{self} is Flying!"
        end
end

class SupermanWrapper

  def self.getInstance()
            # Lazily initialize
            @@superman ||= Superman.new
                        return @@superman
  end

  private_class_method :new
end

threads = Array.new

5.times do
        threads << Thread.new do
        SupermanWrapper.getInstance().fly()
        end
end

threads.each(&:join)
```

In the above example, the snippet `@@superman ||= Superman.new` means if `@@superman` is nil then evaluate the expression on the right of the equality operator and assign it to `@@superman` otherwise, leave it alone. This allows us to delay initializing the object until the first time the `getInstance()` method is invoked. Note that the operator `||=` isn't atomic. The variable `@@superman` may get initialized more than once in a multithreaded environment. We can fix the situation by guarding the initialization in a mutex block as shown in the next snippet. Locking is expensive but necessary to avoid race conditions. Also, this pattern would make sense if the singleton is expensive to create or takes up a lot of resources once created, else instead of initializing the mutex, we could just as well eagerly initialize the singleton.

Also note that, to make our example work, we created a `SupermanWrapper` class to hold the `Superman` object. The reason for the wrapper class is we can't invoke the `new` method if we mark it private, so we need a wrapper over the original class, we intend to make singleton.

```
class Superman
        def fly
        puts "Superman object #{self} is Flying!"
        end
end

class SupermanWrapper
  @@mutex = Mutex.new

  def self.getInstance()
      @@mutex.synchronize {
              # Lazily initialize
              @@superman ||= Superman.new
      }
                      return @@superman
  end

  private_class_method :new
end

threads = Array.new

5.times do
        threads << Thread.new do
        SupermanWrapper.getInstance().fly()
        end
end

threads.each(&:join)
```




We could get around the restriction of invoking `new` once it is marked private by using the `send()` method as follows:

```
class Superman

        @@mutex = Mutex.new

  def self.getInstance()
                      @@mutex.synchronize {
                # Lazily initialize
                @@superman ||= Superman.send :new
                        }
                        return @@superman
  end

        def fly
        puts "Superman object #{self} is Flying!"
```

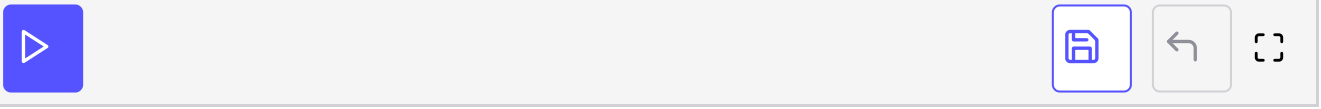
```
        end

  private_class_method :new
end

threads = Array.new

5.times do
        threads << Thread.new do
        Superman.getInstance().fly()
        end
end

threads.each(&:join)
```



The problem with the above approach is that it bypasses accessibility checks when using `send()` which may be an eye-sore to purists.

Another problem with the above approaches is we can subclass the singleton class. Subclassing allows us to have two instances with the same functionality. Languages such as C# (uses sealed) and Java (uses final) provide language-level constructs to block a class from being subclassed. In Ruby if we can subclass a singleton, we can have two instances with same the functionality. E.g. a registry service is an ideal candidate to be implemented using the singleton pattern. However, a malicious user can create another instance of the registry class by subclassing.

Creating a Module

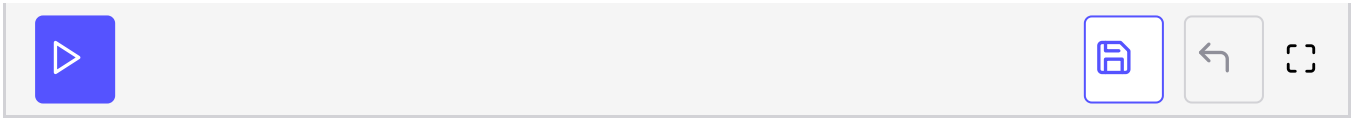
Module is a collection of methods and constants that can't be instantiated, i.e. the `new` method is absent for a module. When a class includes a module, all the module-level methods within that module become class methods in the subject class. Methods defined at the mixin level in the module become instance methods on the instances of the including class. If a module is extended by a class then the methods defined at the mixin level become class methods on the extending class.

We can define a singleton as a module with only module-level methods. Doing so buys us the inability to instantiate and the inability to subclass for free. However, the negative of this approach is that the module can be mixed in with other types.

The example implementation appears below:

```
module Superman
        def self.fly
                puts "Fly!"
        end
end

Superman.fly()
```



Singleton with State

We worked with an example singleton class that had no state i.e. no instance variables. Our focus was on creating a thread-safe singleton class. However, if a singleton has state, then the state must be appropriately protected with locks.