

# Programming for Everybody

## 10. Modules & class methods



# Private and public class methods

Ruby methods define the behaviour of Class objects / instances

By default, methods are **public**, meaning they can be accessed from anywhere in the program

However, it may be useful to define some of them as **private**, when you want to prevent a method from being called from outside the Class definition

# Public methods

**public methods** can be called from outside of the Class definition, on instances of that Class or its subclasses

```
class Animal
  def initialize(name)
    @name = name
  end

  def speak
    "Meow!"
  end
end
```

```
cat = Animal.new("Garfield")
puts cat.speak
```

```
# prints out: Meow!
```

the "cat" instance managed to access the .speak method from within the Animal Class definition scope

# Private methods

**private methods** are preceded by the word **private**; they're for internal usage within the defining Class

They **cannot** be called directly on an instance of the Class

The only way to have external access to a **private** method, is to call it from within a public method

# Private methods

```
class Animal
  def initialize(name)
    @name = name
  end

  private

  def speak
    "Meow!"
  end
end

cat = Animal.new("Garfield")
puts cat.speak
```



```
class Animal
  def initialize(name)
    @name = name
  end

  def access_speak
    speak
  end

  private

  def speak
    "Meow!"
  end
end

cat = Animal.new("Garfield")
puts cat.access_speak
```



# Accessing attributes

If we want to access the instance variables in order to *read* their values, we can do it in two ways. Either by:

1. Defining a method, which will simply return the value of said instance variable
2. By using an attribute reader with this syntax: **attr\_reader**.  
We use this shortcut to *read* (or *get*) the value of an instance variable; that is why it is also called a *getter*

# Accessing attributes (cont.)

```
class Animal
  def initialize(name)
    @name = name
  end

  def name
    @name
  end
end

cat = Animal.new("Garfield")
puts cat.name

# prints out: Garfield
```

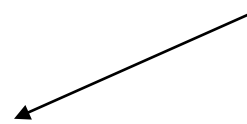
```
class Animal
  attr_reader :name

  def initialize(name)
    @name = name
  end
end

cat = Animal.new("Garfield")
puts cat.name

# prints out: Garfield
```

passing our instance variables as symbols to an **attr\_reader** is the more “Rubyist” way of making them available to be read



# Accessing attributes (cont.)

If we want to access the instance variables in order to ***change*** their values, we can do it in two ways. Either by:

1. Defining a method, which will assign a new value to said instance variable
2. By using an attribute writer with this syntax: **attr\_writer**. We use this shortcut to ***change*** (or *set*) the value of an instance variable; that is why it is also called a *setter*



# Accessing attributes (cont.)

```
class Animal
  def initialize(name)
    @name = name
  end

  def name=(new_name)
    @name = new_name
  end
end

cat = Animal.new("Garfield")
cat.name = "Kitty"
puts cat.name

# prints out: Kitty
```

```
class Animal
  attr_writer :name

  def initialize(name)
    @name = name
  end
end

cat = Animal.new("Garfield")
cat.name = "Kitty"
puts cat.name

# prints out: Kitty
```

# Accessing attributes (cont.)

An attribute accessor, with the syntax **attr\_accessor**, is a shortcut that allows us access to both *read* and *change* the value of an instance variable; it is both a *getter* and a *setter*

```
class Animal
  attr_accessor :name

  def initialize(name)
    @name = name
  end
end

cat = Animal.new("Garfield")
puts cat.name
# prints out: Garfield

cat.name = "Kitty"
puts cat.name
# prints out: Kitty
```

# Modules

**modules** store methods which can then be shared between Classes, allowing us to keep our code DRY

Like Classes, modules also hold methods, but they can't be instantiated -> we can't create objects from a module

Modules are useful if we have methods that we want to reuse in different Classes, while keeping them in a central place to avoid repeating them everywhere

Ruby has some built in modules (ex: Date) which we can use by first using the **require** keyword, followed by their name:

```
require 'date'
```

But we can also create our own

# Modules (cont.)

The module syntax is similar to that of a Class, however modules don't include variables since they, by definition, are mutable while a module is supposed to be immutable

```
module Cream
  def cream?
    true
  end
end
```

The same module can be *mixed* into different Classes in two ways: at *instance level* (through the ***include*** keyword) and at *class level* (through the ***extend*** keyword)

# Modules (cont.)

## Extending a module at instance level

```
module Cream
  def cream?
    true
  end
end
```

```
class Cookie
  include Cream
end
```

```
class Cake
  include Cream
end
```

```
cookie = Cookie.new
p cookie.cream?
```

# prints out: true

```
cake = Cake.new
p cake.cream?
```

# prints out: true

# Modules (cont.)

## Extending a module at Class level

```
module ID
  def item_category(category)
    "You've created a '#{category}' category!"
  end
end

class Cocktail
  extend ID
end

class Cake
  extend ID
end

puts Cocktail.item_category("Cocktail")
# prints: You've created a 'Cocktail' category!

puts Cake.item_category("Cake")
# prints: You've created a 'Cake' category!
```

**Thank** **you.**

