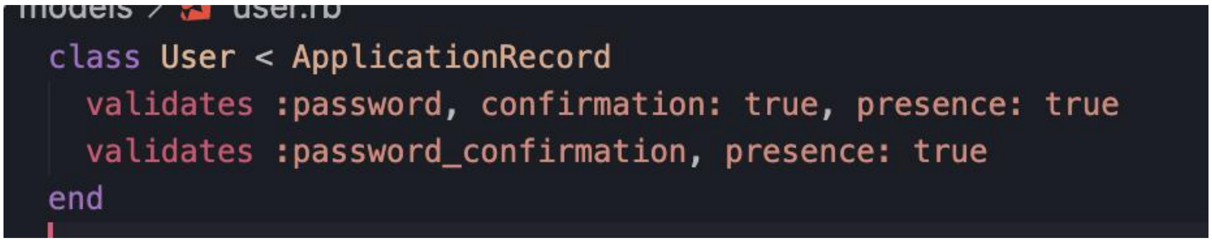
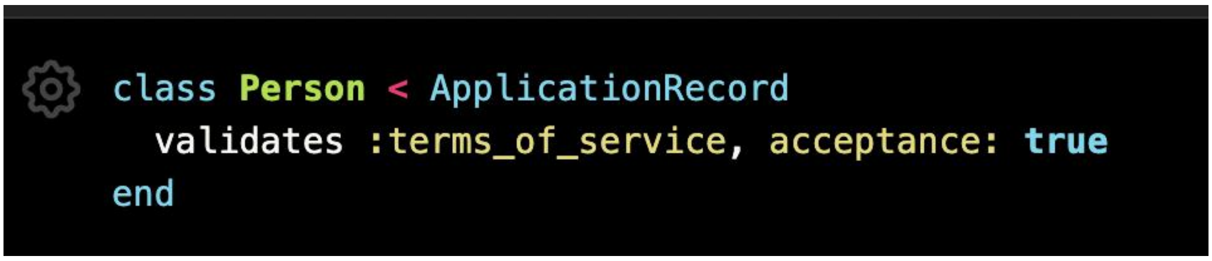
Validations and relationships


models > user.rb

```ruby
class User < ApplicationRecord
  validates :password, confirmation: true, presence: true
  validates :password_confirmation, presence: true
end
```

The model expects to receive a [field] and [field]_confirmation values, then it is going to check if they are the same.
It is used to verify that the user introduced the same password when signing up.

<u>With acceptance</u>
You need presence true so it cannot accept nil values.

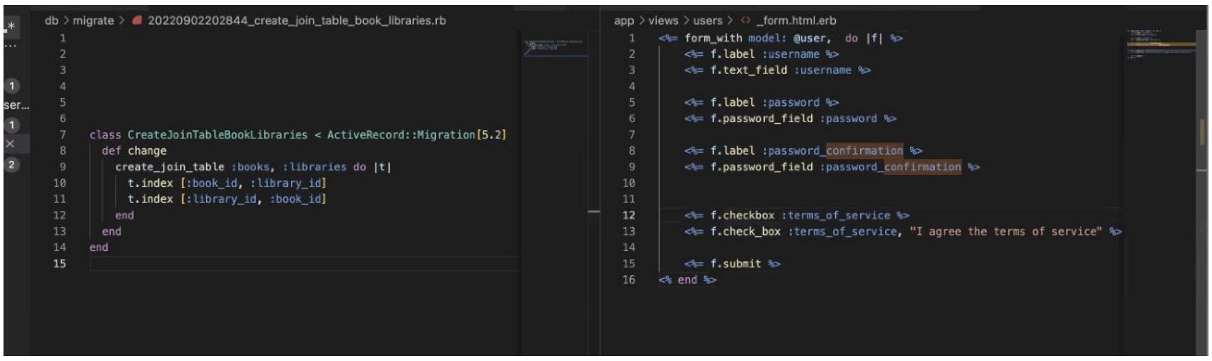


```ruby
class Person < ApplicationRecord
  validates :terms_of_service, acceptance: true
end
```

Useful for form submission. No value is saved into the db.

Create join table

“Create join table” and the models in plural.
Creates a table with no id and columns with no nil.


db > migrate > 20220902202844_create_join_table_book_libraries.rb

```ruby
class CreateJoinTableBookLibraries < ActiveRecord::Migration[5.2]
  def change
    create_join_table :books, :libraries do |t|
      t.index [:book_id, :library_id]
      t.index [:library_id, :book_id]
    end
  end
end
```

app > views > users > _form.html.erb

```erb
<%= form_with model: @user,  do |f| %>
    <%= f.label :username %>
    <%= f.text_field :username %>

    <%= f.label :password %>
    <%= f.password_field :password %>

    <%= f.label :password_confirmation %>
    <%= f.password_field :password_confirmation %>


    <%= f.checkbox :terms_of_service %>
    <%= f.check_box :terms_of_service, "I agree the terms of service" %>

    <%= f.submit %>
<% end %>
```

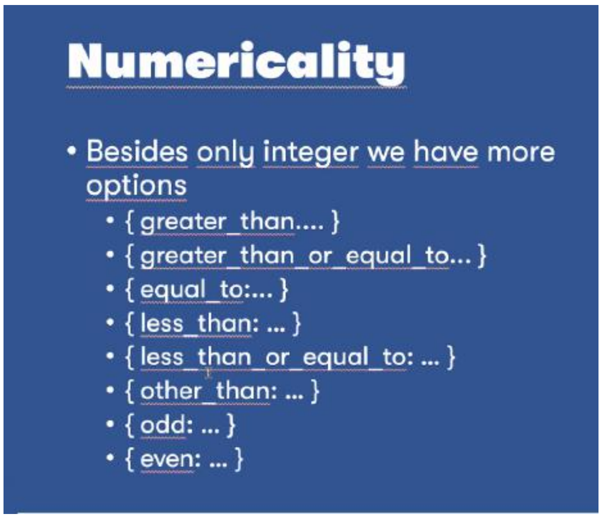
String length validation

```
class Person < ApplicationRecord
  validates :name, length: { minimum: 2 }
  validates :bio, length: { maximum: 500 }
  validates :password, length: { in: 6..20 }
  validates :registration_number, length: { is: 6 }
end
```

Numeric validations

Numerically:
Check if it is a number.

```
class Player < ApplicationRecord
  validates :points, numericality: true
  validates :games_played, numericality: { only_integer: true }
end
```



Numericality

- Besides only integer we have more options
  - { greater_than.... }
  - { greater_than_or_equal_to... }
  - { equal_to:... }
  - { less_than: ... }
  - { less_than_or_equal_to: ... }
  - { other_than: ... }
  - { odd: ... }
  - { even: ... }

Validating presence

Presence: true
Makes the field unable to be nil.

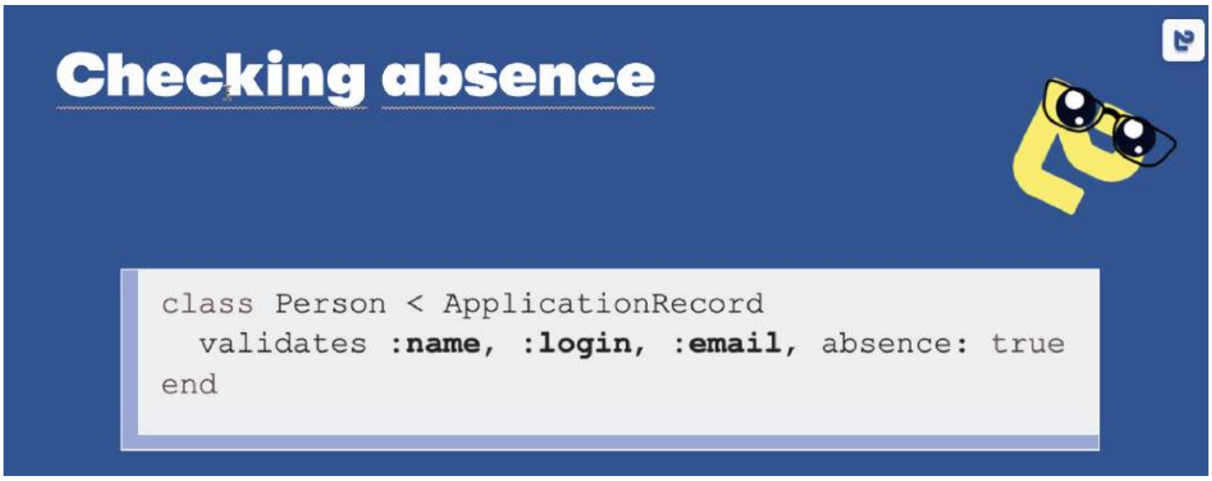
Inclusion and exclusion

Only allows or exclude specific values.

```
validates :boolean_field_name, inclusion: { in: [true, false] }
validates :boolean_field_name, exclusion: { in: [nil] }
```

Checking absence

Almost not used
Validates that the field is empty or nil.

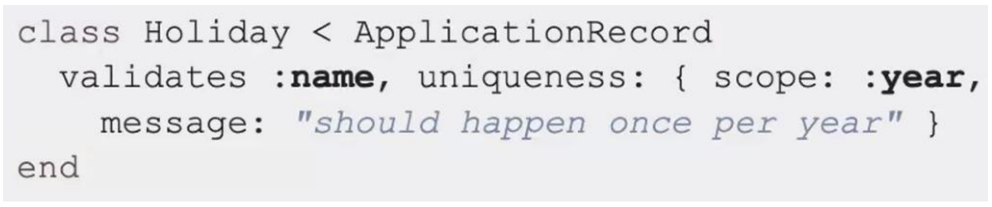


Checking absence

```
class Person < ApplicationRecord
  validates :name, :login, :email, absence: true
end
```

Uniqueness

Only one value per row. You can configure it as well so only one value per another column value.

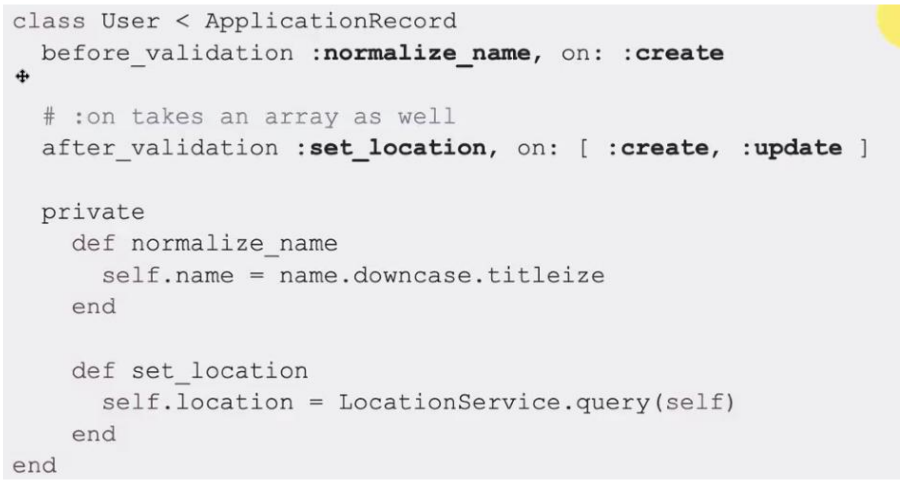


```
class Holiday < ApplicationRecord
  validates :name, uniqueness: { scope: :year,
    message: "should happen once per year" }
end
```

Example: In a year you can only have a holiday that can be called the same.

Callbacks

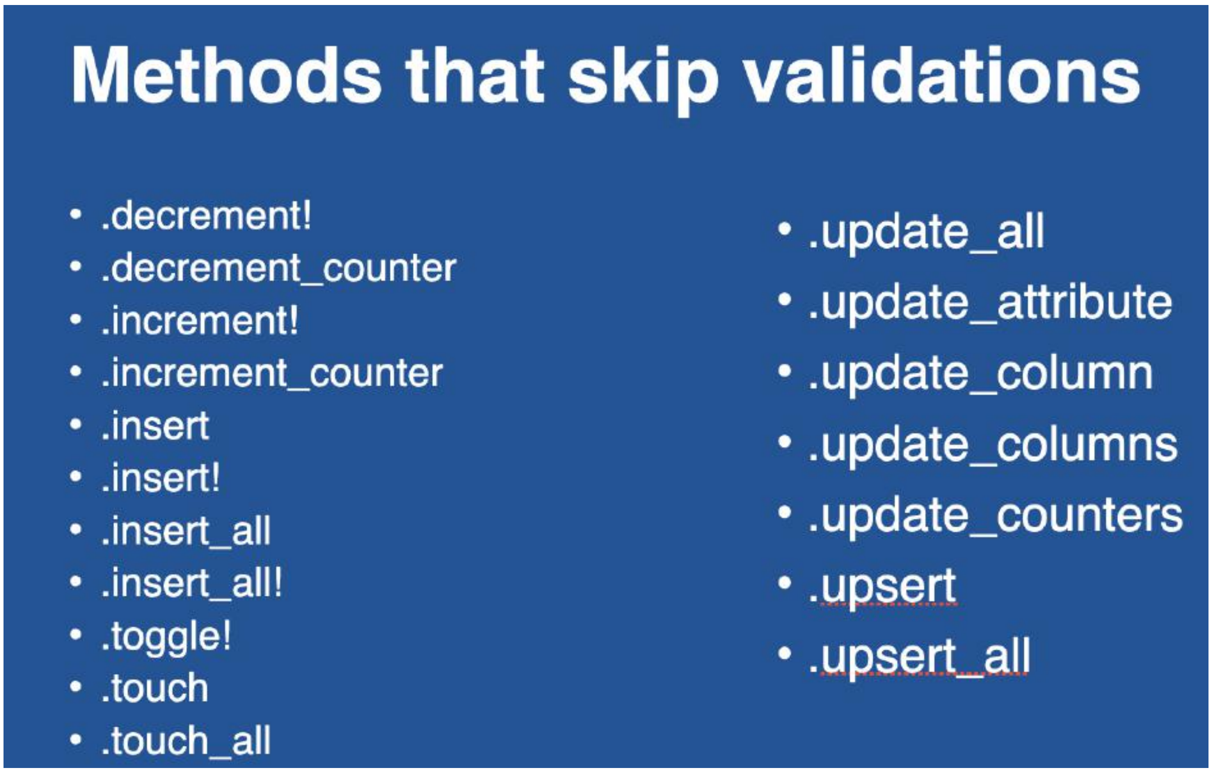
Methods that are called after certain action:



```
class User < ApplicationRecord
  before_validation :normalize_name, on: :create

  # :on takes an array as well
  after_validation :set_location, on: [ :create, :update ]

  private
    def normalize_name
      self.name = name.downcase.titleize
    end

    def set_location
      self.location = LocationService.query(self)
    end
end
```

You usually declare callbacks private.
‘Save’ is called after ‘validation’

There are some methods that avoids callbacks and validations



Methods that skip validations

- .decrement!
- .decrement_counter
- .increment!
- .increment_counter
- .insert
- .insert!
- .insert_all
- .insert_all!
- .toggle!
- .touch
- .touch_all
- .update_all
- .update_attribute
- .update_column
- .update_columns
- .update_counters
- .upsert
- .upsert_all

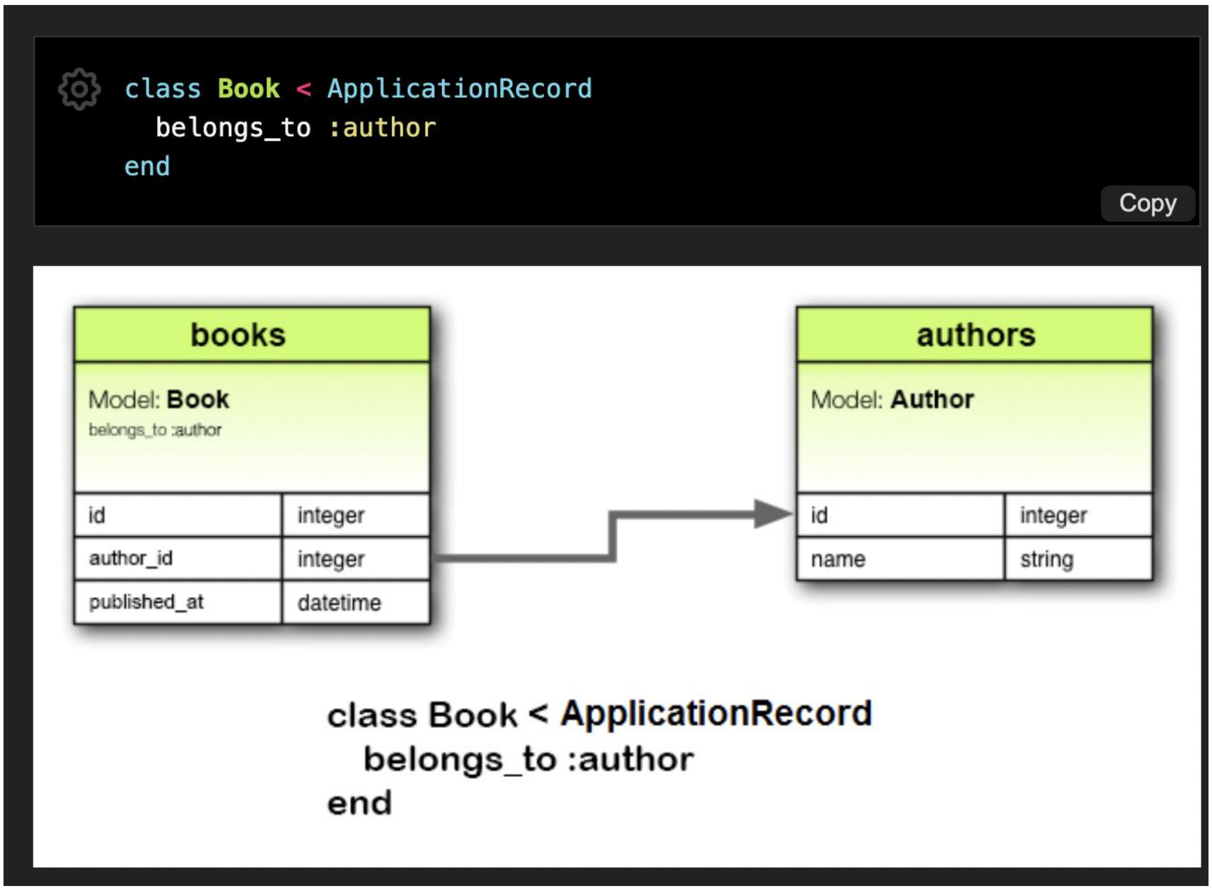
Optional callbacks

Add if or unless at the end of the statement to make them conditional

```
class User < ApplicationRecord
  validates :name, presence: true, if: :admin?

  def admin?
    conditional here that returns boolean value
  end
end
```

Associations

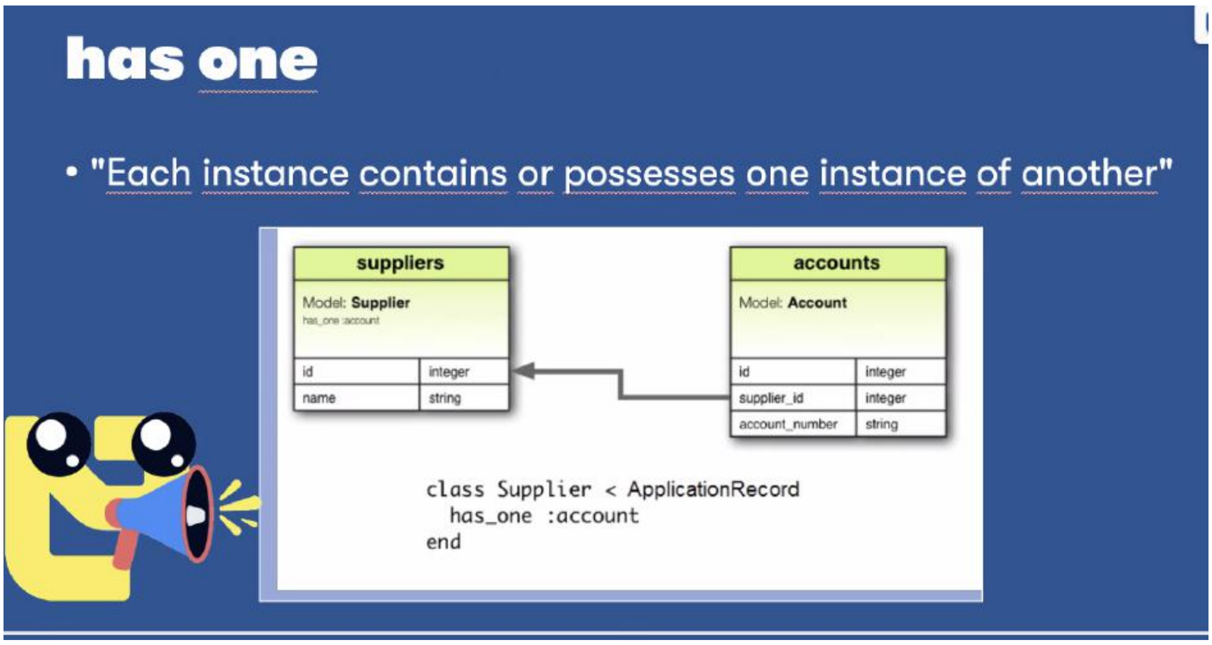
belongs_to



```
class Book < ApplicationRecord
  belongs_to :author
end
```

```
class Book < ApplicationRecord
  belongs_to :author
end
```

Has_one

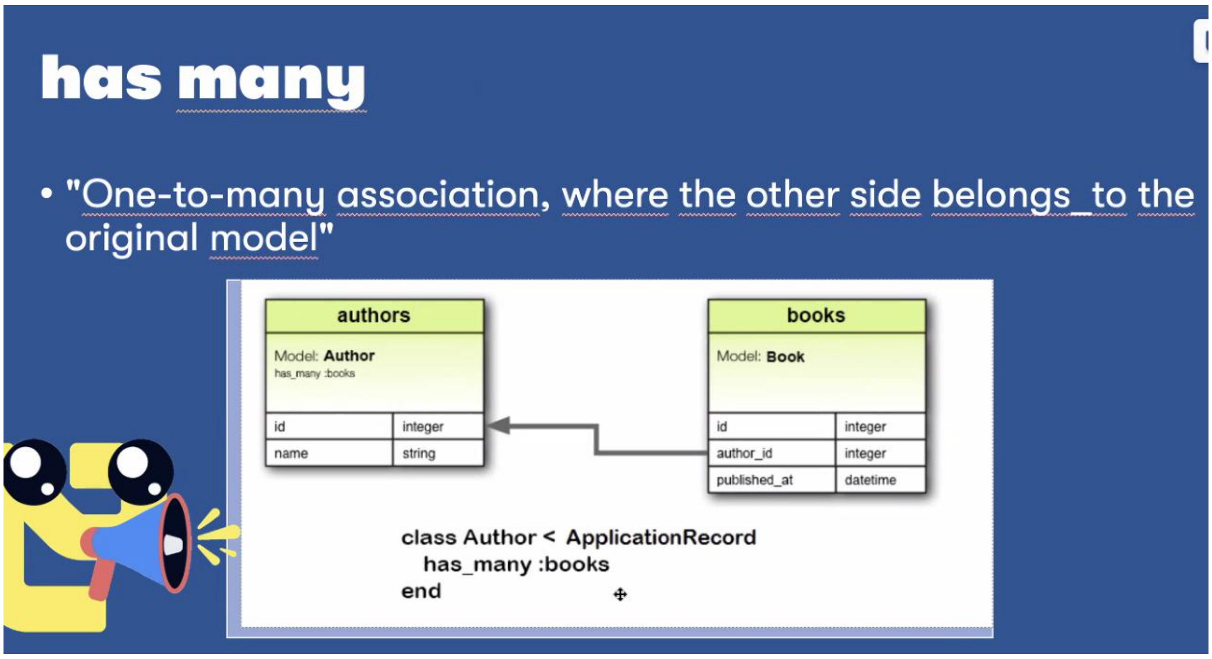


has one

- "Each instance contains or possesses one instance of another"

```
class Supplier < ApplicationRecord
  has_one :account
end
```

Has_many



has many

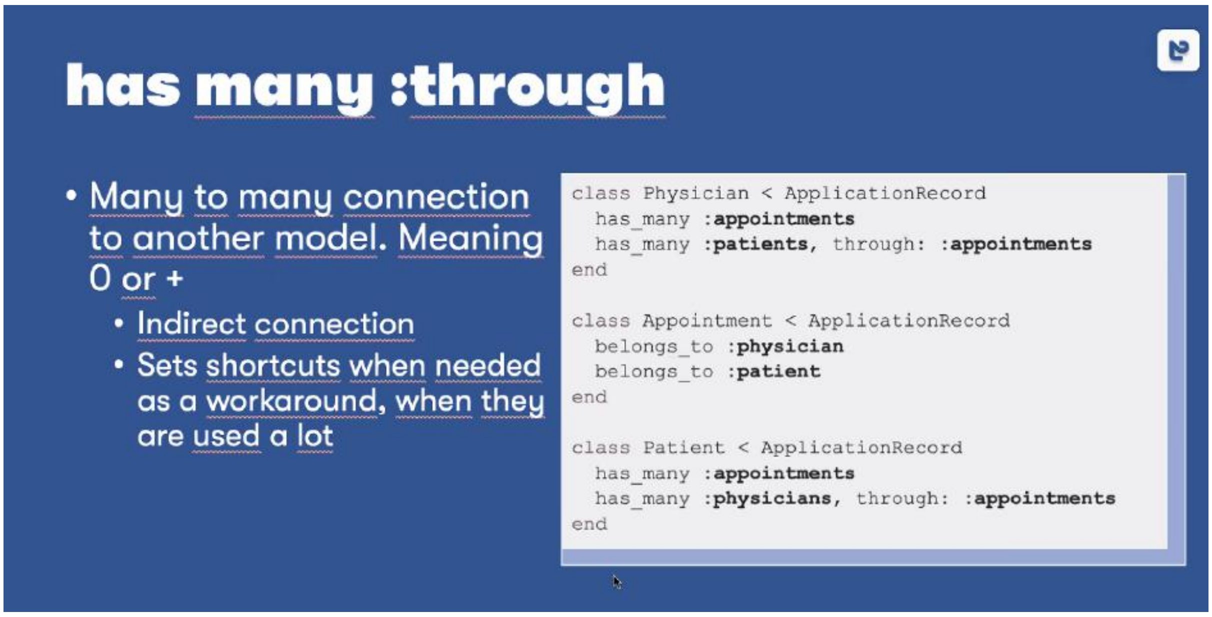
- "One-to-many association, where the other side belongs_to the original model"

authors	
Model: Author has_many :books	
id	integer
name	string

books	
Model: Book	
id	integer
author_id	integer
published_at	datetime

```
class Author < ApplicationRecord
  has_many :books
end
```

Has_many_throught



has many :through

- Many to many connection to another model. Meaning 0 or +
  - Indirect connection
  - Sets shortcuts when needed as a workaround, when they are used a lot

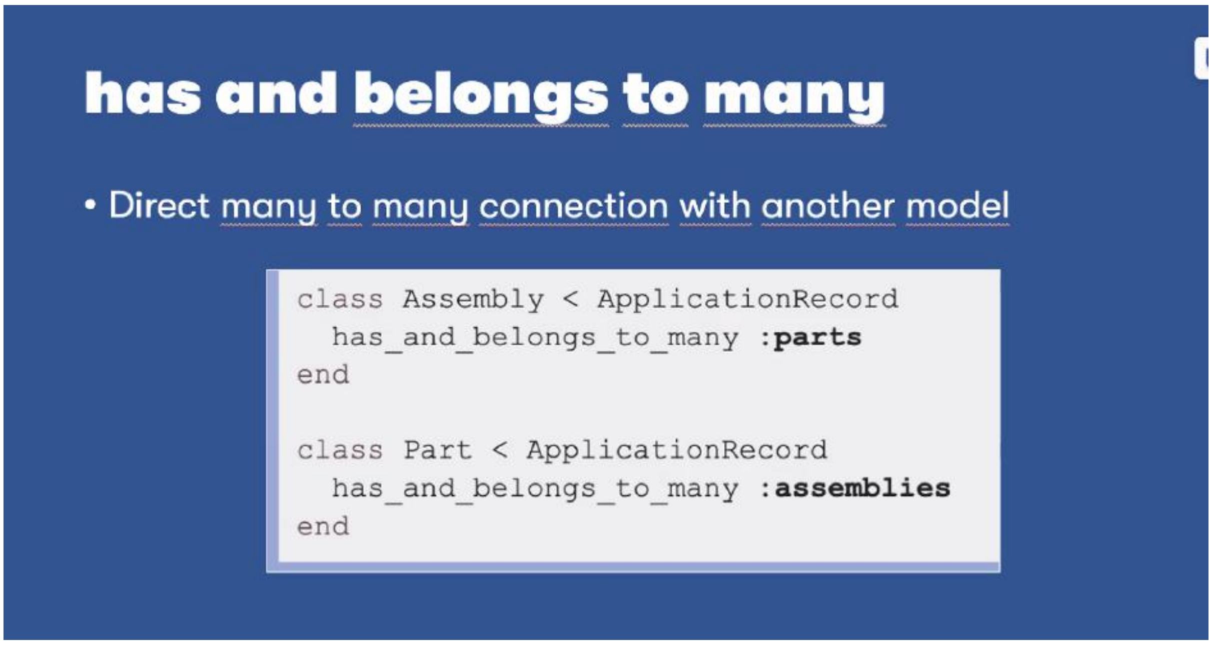
```
class Physician < ApplicationRecord
  has_many :appointments
  has_many :patients, through: :appointments
end

class Appointment < ApplicationRecord
  belongs_to :physician
  belongs_to :patient
end

class Patient < ApplicationRecord
  has_many :appointments
  has_many :physicians, through: :appointments
end
```

has_one_throught

Similar to has_may_through, but only one association per row.

has_and_belongs_to_many

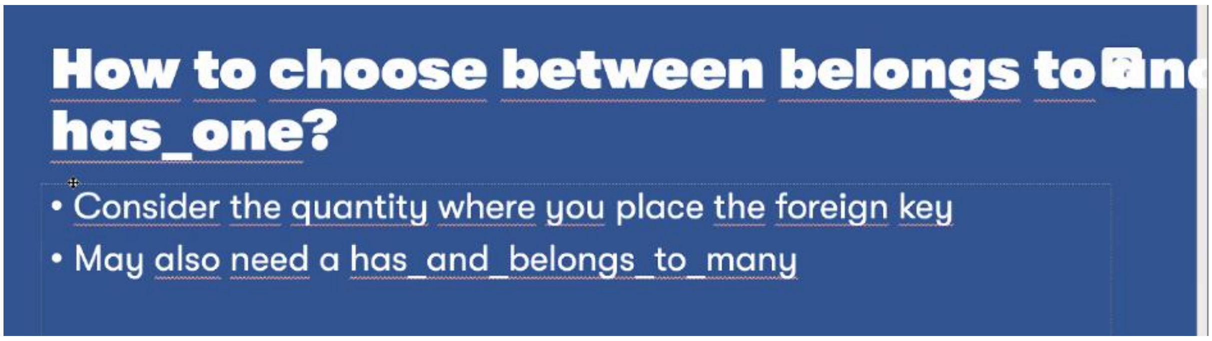
Not recommended to use, similar to has_many_through



has and belongs to many

- Direct many to many connection with another model

```
class Assembly < ApplicationRecord
  has_and_belongs_to_many :parts
end

class Part < ApplicationRecord
  has_and_belongs_to_many :assemblies
end
```

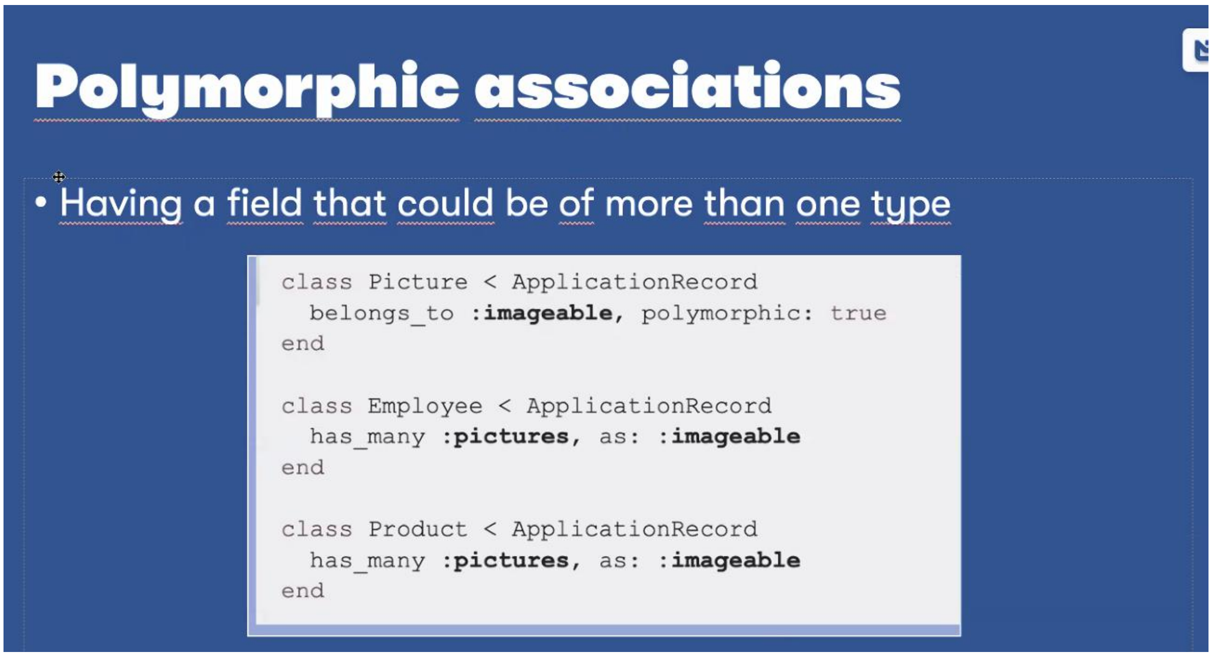


How to choose between belongs to an has_one?

- Consider the quantity where you place the foreign key
- May also need a has_and_belongs_to_many

Polymorphic associations

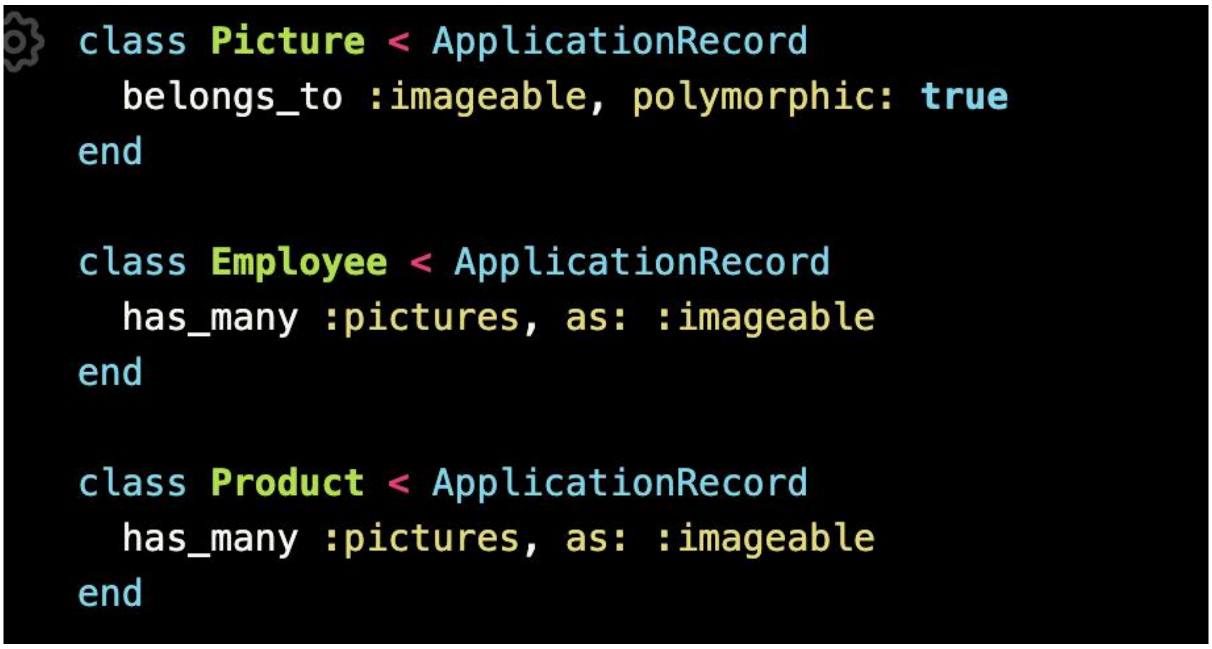
It is when the same field can be used as an association to different models.



Polymorphic associations

- Having a field that could be of more than one type

```ruby
class Picture < ApplicationRecord
  belongs_to :imageable, polymorphic: true
end

class Employee < ApplicationRecord
  has_many :pictures, as: :imageable
end

class Product < ApplicationRecord
  has_many :pictures, as: :imageable
end
```



```ruby
class Picture < ApplicationRecord
  belongs_to :imageable, polymorphic: true
end

class Employee < ApplicationRecord
  has_many :pictures, as: :imageable
end

class Product < ApplicationRecord
  has_many :pictures, as: :imageable
end
```

Useful resources

https://zoom.us/j/95051964684?pwd=eUxXcjk0U2JDTlorNW4rU1F2TTlUdz09
https://juliannaseiki.medium.com/rails-callbacks-cheat-sheet-824295a1a14d