RAILS DAY 07 AK

Associations
an *association* is a connection between two Active Record models

- **belongs_to**

A belongs_to association sets up a one-to-one connection with another model, such that each instance of the declaring model "belongs to" one instance of the other model.

- **has_one**

A has_one association also sets up a one-to-one connection with another model, but with somewhat different semantics (and consequences). This association indicates that each instance of a model contains or possesses one instance of another model. For example, if each supplier in your application has only one account

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

- **has_many**

A has_many association indicates a one-to-many connection with another model. You'll often find this association on the "other side" of a belongs_to association. This association indicates that each instance of the model has zero or more instances of another model.

- **has_many:through**

A has_many :through association is often used to set up a many-to-many connection with another model. This association indicates that the declaring model can be matched with zero or more instances of another model by proceeding *through* a third model

- **has_one:through**

A has_one :through association sets up a one-to-one connection with another model. This association indicates that the declaring model can be matched with one instance of another model by proceeding *through* a third model.

- **has_and_belongs_to_many**

A has_and_belongs_to_many association creates a direct many-to-many connection with another model, with no intervening model.

- **Choosing between belongs_to and has_one**

to set up a one-to-one relationship between two models, you'll need to add belongs_to to one, and has_one to the other
The distinction is in where you place the foreign key (it goes on the table for the class declaring the belongs_to association)

- **Choosing between has_many:through and has_and_belongs_to_many**

has_many —> in through model ,we attributes in addition to the foreign keys of other models
has_and_belongs_to_money —> we add a table in database with foreign keys of two associated tables
The simplest rule of thumb is that you should set up a has_many :through relationship if you need to work with the relationship model as an independent entity. If you don't need to do anything with the relationship model, it may be simpler to set up a has_and_belongs_to_many relationship
You should use has_many :through if you need validations, callbacks or extra attributes on the join model.
———————————

Do we need extra info in join ?   Yes
Join as its own model ?   Yes

Then use has_many :through   ==== more flexible

has_and_belongs_to_many ===== we need to setup join table having foreign keys of both models and can't have extra info in that
——————————

Find through associatios:
Project——has_many:tasks
Task—— belongs_to :project

<u>projects_controller</u>
@project=Project.find(params[:id])
@tasks=Task.find(:all,:conditions=>['project_id=? AND complete=?', @project.id , false])
Or
@tasks=@project.tasks.find(:all,:conditions=>['complete=?', false])
Or
@tasks = @project.tasks.find_all_by_complete(false)
———————————

Polymorphic Associations:
http://railscasts.com/episodes/154-polymorphic-association?autoplay=true

——————————

Polymorphic Associations
With polymorphic associations, a model can belong to more than one other model, on a single association.
To make this work, you need to declare both a foreign key column and a type column in the model that declares the polymorphic interface

Self Joins
Used on a model which has a relationship to itself
In your migrations/schema, you will add a references column to the model itself.

Tips, Tricks, Warnings

Controlling Cache
Most recent query in stored in cache. Reload method is used to discard cache
For belongs_to associations you need to create foreign keys, and for has_and_belongs_to_many associations you need to create the appropriate join table.
Controlling association scope
Associations look for objects in the same module's scope. To associate a model with a model in a different namespace, we have to specify the complete class name in the association definition

Bi-directional Associations (when there is a has_many and belongs_to relation)
Active Record will attempt to automatically identify that two models share a bi-directional association based on the association name. However, Active Record will not automatically identify bi-directional associations that contain a scope or any of the following options:
- :through
- :foreign_key

In case these options are present, Active Record provides the :inverse_of option so you can explicitly declare bi-directional associations

Detailed Association Reference
In database terms, belongs_to association says that this class contains the foreign key.

Methods added by belongs_to
When you declare a belongs_to association, the declaring class automatically gains 6 methods related to the association
Association
The association method returns the associated object, if any. If no associated object is found, it returns nil.
@author = **@book**.author
association=(associate)
The association= method assigns an associated object to this
@book.author = @author
build_association(attributes = {})
The build_association method returns a new object of the associated type. This object will be instantiated from the passed attributes, and the link through this object's foreign key will be set, but the associated object will *not* yet be saved.
@author = **@book**.build_author(author_number: 123,author_name: *"John Doe"*)

create_association(attributes = {})
Same as build association but the object be saved using this
create_association!(attributes = {})
Same as create association but raises ActiveRecord::RecordInvalid if the record is invalid.

Options for belong_to
:autosave
If you set the :autosave option to true, Rails will save any loaded association members and destroy members that are marked for destruction whenever you save the parent object. if the :autosave option is not present, then new associated objects will be saved, but updated associated objects will not be saved.
:class_name
Used to supply the model name
class Book < ApplicationRecord
  belongs_to **:author**, class_name: *"Patron"*
end
:counter_cache
Setting counter_cache : true will query cache to get the data
you would need to add a column named books_count to the model with has_many association. You can override the default column name by specifying a custom column name in the counter_cache declaration instead of true.
:dependent
If you set the :dependent option to:
- :destroy, when the object is destroyed, destroy will be called on its associated objects.
- :delete, when the object is destroyed, all its associated objects will be deleted directly from the database without calling their destroy method.

:foreign_key
The :foreign_key option lets you set the name of the foreign key directly
:primary_key
The :primary_key option allows you to specify a different column instead of default id column
:inverse_of
The :inverse_of option specifies the name of the has_many or has_one association that is the inverse of this association.
:touch
If you set the :touch option to true, then the updated_at or updated_on timestamp on the associated object will be set to the current time whenever this object is saved or destroyed
:validate
If you set the :validate option to true, then associated objects will be validated whenever you save this object. By default, this is false: associated objects will not be validated when this object is saved.
:optional

If you set the :optional option to true, then the presence of the associated object won't be validated. By default, this option is set to false.
Scopes for belongs_to
Where
The where method lets you specify the conditions that the associated object must meet.

```
class Book < ApplicationRecord
  belongs_to :author, -> { where active: true }
end
```

Includes
You can use the includes method to specify second-order associations that should be eager-loaded when this association is used.
Do Any Associated Objects Exist?
You can see if any associated objects exist by using the association.nil? method
:as
Setting the :as option indicates that this is a polymorphic association.

has_one :dependent
Controls what happens to the associated object when its owner is destroyed:

- :destroy causes the associated object to also be destroyed
- :delete causes the associated object to be deleted directly from the database (so callbacks will not execute)
- :nullify causes the foreign key to be set to NULL. Callbacks are not executed.
- :restrict_with_exception causes an exception to be raised if there is an associated record
- :restrict_with_error causes an error to be added to the owner if there is an associated object

:source
To specify a name for :through association

:source_type
To specify a name for has_one association

When are objects saved
When you assign an object to a belongs_to association, that object is not automatically saved.
When you assign an object to a has_one association, that object is automatically saved. In addition, any object being replaced is also automatically saved, because its foreign key will change too.

has_many Association Reference

- collection

The collection method returns a Relation of all of the associated objects.

- collection<<(object, ...)

Adds objects of the related model

- collection.delete(object, ...)

The collection.delete method removes one or more objects from the collection by setting their foreign keys to NULL.

- collection.destroy(object, ...)

The collection.destroy method removes one or more objects from the collection by running destroy on each object.

- collection=(objects)

The collection= method makes the collection contain only the supplied objects, by adding and deleting as appropriate.

- collection_singular_ids

The collection_singular_ids method returns an array of the ids of the objects in the collection.

- collection.clear

The collection.clear method removes all objects from the collection according to the strategy specified by the dependent option. If no option is given, it follows the default strategy. The default strategy for has_many :through associations is delete_all, and for has_many associations is to set the foreign keys to NULL.

- collection.empty?

The collection.empty? method returns true if the collection does not contain any associated objects.

- collection.size

The collection.size method returns the number of objects in the collection.

- collection.find(...)

The collection.find method finds objects within the collection.

- collection.where(...)

The collection.where method finds objects within the collection based on the conditions supplied but the objects are loaded lazily meaning that the database is queried only when the object(s) are accessed.

- collection.exists?(...)

The collection.exists? method checks whether an object meeting the supplied conditions exists in the collection.

Scopes for has_many

- **Extending**

To extend modules

- **group**

The group method supplies an attribute name to group the result set by, using a GROUP BY clause in the finder SQL

- **limit**

The limit method lets you restrict the total number of objects that will be fetched through an association.

- **offset**

The offset method lets you specify the starting offset for fetching objects via an association. For example, -> { offset(11) } will skip the first 11 records.

- **order**

The order method dictates the order in which associated objects will be received (in the syntax used by an SQL ORDER BY clause).

- **readonly**

If you use the readonly method, then the associated objects will be read-only when retrieved via the association.

- **distinct vs uniq**

using .distinct, the program goes through the users array and loads each distinct :role option into an array. When I use .uniq, the program goes through the users array and loads all the :roles into an array, then loops back through that array to eliminate duplicates

```
users.distinct.pluck(:role)
# ["admin", "reader"]
users.pluck(:role).uniq
# ["admin", "admin", "admin", "reader", "reader"] => ["admin", "reader"]
```

Options for has_and_belongs_to_many

- **:join_table**

If the default name of the join table, based on lexical ordering, is not what you want, you can use the :join_table option to override the default.

- **:validate**

If you set the :validate option to false, then associated objects will not be validated whenever you save this object. By default, this is true: associated objects will be validated when this object is saved.

Association Callbacks

Association callbacks are similar to normal callbacks, but they are triggered by events in the life cycle of a collection.

- before_add
- after_add
- before_remove
- after_remove

If a before_add callback throws an exception, the object does not get added to the collection. Similarly, if a before_remove callback throws an exception, the object does not get removed from the collection.

Association Extensions

- proxy_association.owner returns the object that the association is a part of.
- proxy_association.reflection returns the reflection object that describes the association.
- proxy_association.target returns the associated object for belongs_to or has_one, or the collection of associated objects for has_many or has_and_belongs_to_many.

Single Table Inheritance

Sometimes, you may want to share fields and behavior between different models. Let's say we have Car, Motorcycle and Bicycle models. We will want to share the color and price fields and some methods for all of them, but having some specific behavior for each, and separated controllers too.

Vehicle Model

```
rails generate model vehicle type:string color:string price:decimal{10.2}
```

To make the car model inherit all associations and public methods of vehicle

```
rails generate model car --parent=Vehicle
```