OOP short

Object Oriented Programming

Encapsulation: Bundling data and methods that operate on it, restricting direct access.

Abstraction: Hiding complex details, presenting a simple interface.

Inheritance: Child classes can inherit methods and properties from parent classes.

Polymorphism: Allows using an object like its parent while maintaining its own unique behavior.

OOP

Encapsulation	private public methods
Abstra-ctions	user blindly
Inheri-tance	u can replace parent methods with child methods
Polymorphism	Polymorphism gives us a way to use an object exactly like its parent but keeping its own methods as they are

Method Access Control short

Public: Methods accessible by anyone.

Protected: Only accessible by the defining class and its subclasses.

Private: Only accessible by the current object; cannot be called with an explicit receiver.

Method Access Control

Public methods can be called by everyone - no access control is enforced. A class's instance methods (these do not belong only to one object; instead, every instance of the class can call them) are public by default; anyone can call them. The initialize method is always private.

Protected methods can be invoked only by objects of the defining class and its subclasses. Access is kept within the family. However, usage of protected is limited.

Private methods cannot be called with an explicit receiver - the receiver is always self. This means that private methods can be called only in the context of the current object; you cannot invoke another object's private methods.

ActiveJob

emails

image processing

external API call

analytics calculations

Rails Enige

```
Rails::Engine allows you to wrap a specific Rails application or subset of functionality and share it with other applications or within a larger packaged application. Every Rails::Application is just an engine, which allows for simple feature and application sharing.
Any Rails: :Engine is also a Rails: :Ra iltie, so the same methods (like rake_tasks and genera tors) and config uration options that are available in railties can also be used in engines.
# lib/my _en gine.rb
module MyEngine
    class Engine < Rails: :Engine
   end
end
```

RESTful

INDEX	GET users
NEW	GET users/new
SHOW	GET users/1
EDIT	GET users/1/edit
UPDATE	PUT/PATCH users/1
DELETE	GET users/1/d-elete
DESTROY	DELETE users/1
CREAET	POST users

How should you test routes

```
# Asserts that the default action is generated for a route with no action
assert _ge nerates " /it ems ", contro - ller: " ite ms", action: " ind ex"
# Tests that the list action is properly routed
assert _ge nerates " /it ems /li st", contro ller: " ite ms", action: " lis t"
# Tests the generation of a route with a parameter
assert _ge nerates " /it ems /li st/ 1", { contro ller: " ite ms", action: " lis t", id: " 1" }
# Asserts that the generated route gives us our custom route
```

How should you test routes (cont)

> assert_generates "changesets/12", { controller: 'scm', action: 'show_diff', revision: "12" }
# Asserts that POSTing to /items will call the create action on ItemsController
assert_recognizes({controller: 'items', action: 'create'}, {path: 'items', method: :post})

Active Record short

ORM: Maps objects to database tables, providing methods to represent models, relationships, and validations.	Associations: Defines relationships like has_many, belongs_to, and has_many :through.

Conventions: Naming conventions for tables and associations, e.g., User has many Posts.

ActiveRecord

Active Record is the M in MVC - the model - which is the layer of the system responsible for representing business data and logic. Active Record facilitates the creation and use of business objects whose data requires persistent storage to a database. It is an implementation of the Active Record pattern which itself is a description of an Object Relational Mapping system.

ActiveRecord (cont)

In Active Record, objects carry both persistent data and behavior which operates on that data. Active Record takes the opinion that ensuring data access logic as part of the object will educate users of that object on how to write to and read from the database.
Active Record gives us several mechanisms, the most important being the ability to:
Represent models and their data.
Represent associations between these models.
Represent inheritance hierarchies through related models.
Validate models before they get persisted to the database.
Perform database operations in an object-oriented fashion.

Object Relational Mapping

Object Relational Mapping, commonly referred to as its abbreviation ORM, is a technique that connects the rich objects of an application to tables in a relational database management system. Using ORM, the properties and relationships of the objects in an application can be easily stored and retrieved from a database without writing SQL statements directly and with less overall database access code.

Active Record conventions

user has_many	posts
Article	articles
PostComment	post_comments
Mouse	mice
post belongs_to	user

Fat controllers

retrieving data from the model, transforming it as appropriate for the view, and then passing it to the view for rendering

Refactor of models

If some code does work from the point of view of an ActiveRecord model, it can go into the model.
If some code does work that spans multiple tables/objects, and doesn't really have a clear owner, it could go into a Service Object.
Anything that's attribute-like (like attributes calculated from associations or other attributes) should go into your ActiveRecord model.
If you have logic that has to orchestrate the saving or updating of multiple models at once, it should go into an ActiveModel Form Object.

Refactor of models (cont)

If code is mostly meant for displaying or formatting models in a view, it should go into a Rails helper or a Presenter.

E-Tags short

Definition: Response headers that identify a resource version, aiding in caching and content validation.	Usage: If-None-Match header checks if a resource has changed, allowing "304 Not Modified" responses.

E-tags

Definition: An E-tag is a response header used to identify a specific version of a resource on a server. It's often a unique identifier, such as a hash or a version number, that changes whenever the resource changes.

E-tags (cont)

Caching: E-tags are crucial for caching mechanisms. When a client requests a resource, it receives an E-tag with the response. On subsequent requests, the client can send this E-tag back to the server with an "If-None-Match" header, allowing the server to determine if the resource has changed. If it hasn't, the server can respond with a "304 Not Modified" status, reducing bandwidth usage by skipping the resource transfer.

Content Validation: E-tags also help ensure content integrity. By comparing E-tags, clients and servers can verify that they are accessing the correct version of a resource, ensuring consistency between versions.

Implementation: In practice, E-tags are generated and managed by the server. Developers might create custom logic to generate these tags or rely on server frameworks that handle them automatically.

E-tags (cont)

E-tags are particularly useful in: Web APIs: For API responses, E-tags help reduce data transfer and ensure clients receive the latest information. Static Content: For web pages, CSS files, and JavaScript assets, E-tags prevent redundant downloads, speeding up page loads.

SOLID short

Single Responsibility: A class should handle one responsibility.	Open/Closed: Classes should be open for extension but closed for modification.
Liskov Substitution: Subtypes should be substitutable for their parent types.	Interface Segregation: Clients shouldn't depend on unused interfaces.

Dependency Inversion: High-level modules should depend on abstractions, not low-level modules.

SOLID

Single Responsibility Principle: A class should have only one reason to change, meaning it should handle a single responsibility or functionality. This helps create modular code that's easier to understand and maintain.

SOLID (cont)

Open/Closed Principle: Software entities (such as classes or functions) should be open for extension but closed for modification. This means you should be able to add new functionality without altering existing code, typically through inheritance or composition.

Liskov Substitution Principle: Subtypes should be substitutable for their base types. In other words, derived classes should be able to replace their parent classes without affecting the functionality of the application.

Interface Segregation Principle: Clients should not be forced to depend on interfaces they do not use. This encourages the creation of small, specific interfaces, rather than large, general ones, making the design more flexible and reducing coupling.

SOLID (cont)

Dependency Inversion Principle: High-level modules should not depend on low-level modules; both should depend on abstractions. This means code should rely on abstract interfaces rather than concrete implementations, promoting decoupling and making the system easier to extend and maintain.

self

The keyword self in Ruby gives you access to the current object – the object that is receiving the current message. To explain: a method call in Ruby is actually the sending of a message to a receiver

Ruby method lookup path

user	User	Parent::	Object
			User

Ruby callbacks

Create	Update
before_save; before_validation	after_update; after_commit
Destroying	**Touch**
around_destroy; after_rollback	after_initialize; after_find

Proc and lambda

Lambda	Proc
Proc class (lambda)	Proc class
checks the number of arguments passed to it	proc does not
lambda returns, it passes control back to the calling method	proc returns, it does so immediately, without going back to the calling method

Asset Pipeline

The asset pipeline provides a framework to concatenate and minify or compress JavaScript and CSS assets. It also adds the ability to write these assets in other languages and pre-processors such as CoffeeScript, Sass, and ERB

Caching short

Page Caching: Caches entire pages directly to disk.	Action Caching: Similar to page caching but runs before filters.

Caching short (cont)

Fragment Caching: Caches parts of views separately.	Low-Level Caching: Allows caching values or query results, reducing database load.

Caching 1/2

Page caching is a Rails mechanism which allows the request for a generated page to be fulfilled by the web server (i.e. Apache or NGINX) without having to go through the entire Rails stack. While this is super fast it can't be applied to every situation (such as pages that need authentication). Also, because the web server is serving a file directly from the filesystem you will need to implement cache expiration.	Page Caching cannot be used for actions that have before filters - for example, pages that require authentication. This is where Action Caching comes in. Action Caching works like Page Caching except the incoming web request hits the Rails stack so that before filters can be run on it before the cache is served. This allows authentication and other restrictions to be run while still serving the result of the output from a cached copy.

Caching 1/2 (cont)

Dynamic web applications usually build pages with a variety of components not all of which have the same caching characteristics. When different parts of the page need to be cached and expired separately you can use Fragment Caching. Fragment Caching allows a fragment of view logic to be wrapped in a cache block and served out of the cache store when the next request comes in.

Caching 2/2

Low-level	SQL caching
Sometimes you need to cache a particular value or query result instead of caching view fragments. Rails' caching mechanism works great for storing any kind of information.	Query caching is a Rails feature that caches the result set returned by each query. If Rails encounters the same query again for that request, it will use the cached result set as opposed to running the query against the database again.

Caching 2/2 (cont)

```
Rails.cache.fetch("#{cache_key_with_version}/competing_price", expires_in: 12.hours) do
  Competitor::API.find_price(id)
end
```

```
User.first;
User.first
```

Migrations

Rails provides a set of rake tasks to work with migrations which boil down to running certain sets of migrations.
The very first migration related rake task you will use will probably be rake db:migrate. In its most basic form it just runs the up or change method for all the migrations that have not yet been run. If there are no such migrations, it exits. It will run these migrations in order based on the date of the migration.

Migrations (cont)

Note that running the db:migrate also invokes the db:schema:dump task, which will update your db/schema.rb file to match the structure of your database.

Associations

belongs_to

has_one

has_many

has_many :through

has_one :through

has_and_belongs_to_many

scopes

Scoping allows you to specify commonly-used queries(it can be considered as a shortcut for long or most frequently used queries) which can be referenced as method calls on the association objects or models. With these scopes, you can use every method previously covered such as where, joins and includes. All scope methods will return an ActiveRecord::Relation object which will allow for further methods (such as other scopes) to be called on it.

Basic info

Class is the blueprint from which individual objects are created

class variables are shared between a class and all its subclasses

Objects are instances of the class.

class instance variables only belong to one specific class

Use a constructor in Ruby

```
def initialize({});end;
```

getter and setter methods in Ruby

attr_reader

attr_accessor

Class and a module

Modules are collections of methods and constants.

They cannot generate instances. Classes may generate instances (objects), and have per-instance state (instance variables).

everything is an object in Ruby

Every data type that we work with is a class and classes are objects. Even the Object class is an object. Strings, integers, floats, hashes, arrays, symbols, classes, modules, errors and more are all objects.

Rack

Rack is the underlying technology behind nearly all of the web frameworks in the Ruby world.
"Rack" is actually a few different things:
An architecture - Rack defines a very simple interface, and any code that conforms
to this interface can be used in a Rack application. This makes it very easy to build small, focused, and reusable bits of code and then use Rack to compose these bits into a larger application.
A Ruby gem - Rack is is distributed as a Ruby gem that provides the glue code needed to compose our code.

```
require "rack"
require "thin"
class HelloWorld
def call(env)
```

Rack (cont)

[200, { "Content-Type" => "text/plain" }, ["Hello World"]]
end
end
Rack::Handler::Thin.run HelloWorld.new
https://github.com/sunlightlabs/rack-honeypot
Rack defines a very simple interface. Rack compliant code must have the following three characteristics:
It must respond to call
The call method must accept a single argument -
This argument is typically called env or environment, and it bundles all of the data about the request.
The call method must return an array of three elements These elements are, in order,
status for the HTTP status code, headers, and body for the actual content of the response.

Rack (cont)

A nice side effect of the call interface is that procs and lambdas can be used as Rack objects.
Middleware are the building blocks of larger applications built using the Rack pattern.
Each middleware is a Rack compatible application, and our final application is built by composing together, or nesting these middleware.
Unlike base Rack apps, middleware must be classes as they need to have an initializer which will be passed the next app in the chain.
For our first middleware example, we'll introduce a middleware that logs the amount of time the request took and adds that to the response.

Rack (cont)

To begin, we'll update our core Rack app to sleep for 3 seconds to give us something worth logging, and then we'll build our middleware:
Rack::Handler::Thin.run LoggingMiddleware.new(app)
Middleware are perfect for non-app specific logic.
Things like setting caching headers, logging, parsing the request object, etc. all are great use cases for Rack middleware.
For example, in Rails, cookie parsing, sessions, and param parsing are all handled by Middleware.

Rack Middleware

```
require "rack"
require " thi n"
app = -> (env) do
    sleep 3
   [ 200, { " Con ten t-T ype " => " tex t/p -lai n" }, ["Hello World \n"] ]
```

Rack Middleware (cont)

> end
class LoggingMiddleware
  def initialize(app)
    @app = app
  end
  def call(env)
    before = Time.now.to_i
    status, headers, body = @app.call(env)
    after = Time.now.to_i
    log_message = "App took #{after - before} seconds."
    [status, headers, body << log_message]
  end
end
Rack::Handler::Thin.run LoggingMiddleware.new(app)

optimistic pessimistic locking

Optimistic	Pessimistic

optimistic pessimistic locking (cont)

Optimistic locking is a mechanism to prevent data overrides by assuming that a database transaction conflict rarely happens

When one user is editing a record and we maintain an exclusive lock on that record, another user is prevented from modifying this record until the lock is released or the transaction is completed. This explicit locking is known as a pessimistic lock

uses a "version-number" column to track changes in each table that needs to implement concurrent access

Optimistic locking is just a mechanism to prevent processes from overwriting changes by another process. Optimistic locking is not a magic wand to manage or auto-merge any conflicting changes. It can only allow users to alert or notify about such conflicting changes.

optimistic pessimistic locking (cont)

Optimistic locking works by just comparing the value of the "version" column. Thus, optimistic locking is not a real database lock

multiple users can read the same resource at the same time but if more then one tries to modify the database , then we prevent it

multiple users will not be able to read while others are reading

Advisory Lock

optimistic pessimistic locking (cont)

An advisory lock is a voluntary locking mechanism that requires transactions to explicitly request and release locks on resources. How it works: Transactions explicitly acquire advisory locks on resources, signaling to other transactions that they should also respect these locks. Advisory locks don't inherently block transactions; instead, they rely on voluntary adherence. Use cases: Advisory locks are useful in systems where developers want fine-grained control over concurrency or when specific business logic dictates locking behavior, such as file management systems.

postgres indexes

B-tree Index: Default type, suitable for equality (=) and range queries (<, >, BETWEEN).

Hash Index: Optimized for equality comparisons (=).

postgres indexes (cont)

GIN Index: Useful for indexing array, JSONB, and full-text search fields.

GiST Index: Flexible structure for complex data types, including geometric and range queries.

Partial Index: create index where condition

The order of columns in a multi--column index affects how well it optimizes queries. Consider these factors: Query Patterns: Place the column most frequently used in queries or filters first. This increases the chances of the index being utilized effectively. Selective Columns: Columns with high selectivity (i.e., a wide range of unique values) should appear first. This maximizes the potential for early filtering. Combining Columns: If queries often filter by a combination of columns, ensure the index order matches the most common query patterns.

Highiest cardinality means better index. Columns with high cardinality have many unique values relative to the total number of rows.

singleton methods, Eigenclass

In Ruby, a class is an object. Indeed, a class is literally an instance of the class Class The eigenclass is an unnamed instance of the class Class attached to an object and which instance methods are used as singleton methods of the defined object.

RubyGems

The RubyGems software allows you to easily download, install, and use ruby software packages on your system.
The software package is called a “gem” which contains a packaged Ruby application or library.

Eigenclass

An eigenclass is a unique, anonymous class associated with an individual object in Ruby. It allows for the addition of methods directly to that object, without affecting other instances of its class.

Every Ruby object has an associated eigenclass, which is created automatically the first time a singleton method (a method defined only for that specific object) is added to it. The eigenclass stores these singleton methods.

Eigenclass (cont)

Eigenclasses can be useful in Rails for extending or modifying specific instances of classes without affecting the class as a whole: Singleton Methods: You might use a singleton method to add behavior directly to an individual object, such as a single instance of a model, without altering its class definition. Meta-Programming: Eigenclasses play a role in meta-programming techniques, allowing for dynamic modifications to objects.

class User < ApplicationRecord end user = User.find(1) def user.greet "Hello, #{self.name}!" end puts user.greet # Outputs "-Hello, [User's name]!"

build your own Ruby gem

.gemspec

lib/mygem

gem build .gemspec

Use the basic lib/gem.rb and lib/gem/ structure for code.

Put any executables in bin, any data files in data and tests in test or spec.

build your own Ruby gem (cont)

Don't require or depend upon files outside of the load path. (VERSION files often seem to live in odd places in gems.)

Do not require 'rubygems'.

Do not tamper with the $LOAD_-PATH.

Favourites gems

rspec	cancancan	Active-ModelS-eri-alizers
capistrano	sidekiq	annotate

Filters in controllers

```
class ChangesController
< ApplicationController
    aro und _action
:wrap_ in_ tra nsa -
ction, only: :show

    private

   def wrap_i n_t ran -
saction
        Act ive Rec -
ord ::B ase.tr ans -
action do
            begin
                yield
            ensure
                raise
Active Rec ord ::R -
ollback
            end
        end
   end
end
```

Strong params

Strong Parameters is a feature of Rails that prevents assigning request parameters to objects unless they have been explicitly permitted. It has its own DSL (Domain Specific Language, or in other words, a predefined syntax it understands), that allows you to indicate what parameters should be allowed. It also lets you indicate if each parameter should be a hash, array or scalar (i.e. integer, string, etc.), as well as some other functionality

Controller specs

Send http requests to application and writing assertions about the response.

yield, content_for

They are opposite ends of the rendering process, with yield specifying where content goes, and content_for specifying what the actual content is.

yield, content_for (cont)

The best practice is to use yield in your layouts, and content_for in your views. There is a special second use for content_for, where you give it no block and it returns the previously rendered content. This is primarily for use in helper methods where yield cannot work. Within your views, the best practice is to stick to yield :my_content to recall the content, and content_for :my_content do...end to render the content.

nested layouts

This is a live content block, but has not yet been populated. Please check back soon.

Security

Content Security Policy	xss inj
https://rails-s-qli.org/	sql inj
use ssl	session hijacking
reset_session (expire)	session fixation

Password

Iterate over an HMAC with a random salt for about a 100ms duration and save the salt with the hash. Use functions such as password_hash, PBKDF2, Bcrypt and similar functions.

https (ssl)

HTTPS protects the communication between your browser and server from being intercepted and tampered with by attackers. This provides confidentiality, integrity and authentication to the vast majority of today's WWW traffic. Any website that shows a lock icon in the address bar is using HTTPS

Unit testing

In computer programming, unit testing is a software testing method by which individual units of source code—sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures—are tested to determine whether they are fit for use

Patterns

Creational	Structural	Behaviour
Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.	Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.	Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Patterns (cont)

Singletone	Decorator	Comma
	is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.	is a behavio design pattern that turn a reque into a stand-- alone object th contains all inform ation about th request. This transfor ation let you parame terize methods with different requests delay or queue a request' executio and support undoabl operat- ions.

Patterns (cont)

Prototype	Facade	Chain of respon sibility

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

What is the primary technique for writing a test

Test actual result of execution, integration tests, rather than only unit tests. Write scenario as a complex test

Eigenclass

An eigenclass is a unique, anonymous class associated with an individual object in Ruby. It allows for the addition of methods directly to that object, without affecting other instances of its class.

Every Ruby object has an associated eigenclass, which is created automatically the first time a singleton method (a method defined only for that specific object) is added to it. The eigenclass stores these singleton methods.

Eigenclass (cont)

Eigenclasses can be useful in Rails for extending or modifying specific instances of classes without affecting the class as a whole: Singleton Methods: You might use a singleton method to add behavior directly to an individual object, such as a single instance of a model, without altering its class definition. Meta-Programming: Eigenclasses play a role in meta-programming techniques, allowing for dynamic modifications to objects.

```
class User < ApplicationRecord end user = User.find(1) def user.greet "Hello, #{self.name}!" end puts user.greet # Outputs "-Hello, [User's name]!"
```